**Esame SO**

Sommario

[CPU (Program Status Word) 3](#_Toc139975265)

[System call 4](#_Toc139975266)

[Stato di un processo e transizioni di stato 4](#_Toc139975267)

[Processi sospesi 5](#_Toc139975268)

[Process Control Block 5](#_Toc139975269)

[Sincronizzazione dei processi 6](#_Toc139975270)

[Race condition 6](#_Toc139975271)

[Sezioni critiche (SC) 6](#_Toc139975272)

[Proprietà di un’implementazione di una sezione critica 7](#_Toc139975273)

[Sincronizzazione di controllo e operazioni indivisibili 7](#_Toc139975274)

[Approcci alla sincronizzazione 7](#_Toc139975275)

[Ciclare vs bloccare 8](#_Toc139975276)

[Supporto HW per la sincronizzazione dei processi 8](#_Toc139975277)

[Istruzione test-and-set 9](#_Toc139975278)

[Istruzione swap 9](#_Toc139975279)

[Introduzione ai problemi di sincronizzazione classici 10](#_Toc139975280)

[Produttori-consumatori con buffer limitato 10](#_Toc139975281)

[Lettori e scrittori 12](#_Toc139975282)

[Filosofi a cena 13](#_Toc139975283)

[Approcci algoritmici per le SC 14](#_Toc139975284)

[Algoritmi a due processi 15](#_Toc139975285)

[Algoritmi a due processi (Algoritmo di Dekker) 16](#_Toc139975286)

[Algoritmi a due processi (Algoritmo di Peterson) 17](#_Toc139975287)

[Algoritmi con n processi 18](#_Toc139975288)

[Algoritmi con n processi (Algoritmo del panettiere) 18](#_Toc139975289)

[Semafori 20](#_Toc139975290)

[Utilizzo dei semafori nei sistemi concorrenti 20](#_Toc139975291)

[Semafori binari 22](#_Toc139975292)

[Produttori-consumatori a buffer singolo con semafori 22](#_Toc139975293)

[Produttori-consumatori a n buffer con semafori 23](#_Toc139975294)

[Lettori-scrittori con soluzione rifinita mediante contatori 24](#_Toc139975295)

[Lettori-scrittori con semafori (preferenza ai lettori) 25](#_Toc139975296)

[Implementazione dei semafori 26](#_Toc139975297)

[Monitor 27](#_Toc139975298)

[Variabili di condizione 28](#_Toc139975299)

[Problema del barbiere addormentato 29](#_Toc139975300)

[Scheduling 30](#_Toc139975301)

[Terminologia e concetti 30](#_Toc139975302)

[Tecniche di scheduling 31](#_Toc139975303)

[Il ruolo delle priorità 32](#_Toc139975304)

[Scheduling senza prelazione 32](#_Toc139975305)

[Scheduling First-Come, First-Served (FCFS) 32](#_Toc139975306)

[Scheduling Shortest Job First (SJF) 33](#_Toc139975307)

[Scheduling High Response Ratio Next (HRN) 35](#_Toc139975308)

[Scheduling con prelazione 36](#_Toc139975309)

[Scheduling Round-Robin con Time-Slice (RR) 36](#_Toc139975310)

[Scheduling Least Completed Next (LCN) 38](#_Toc139975311)

[Scheduling Shortest Time to Go (STG) 39](#_Toc139975312)

[Scheduling in pratica 40](#_Toc139975313)

[Scheduler a lungo, medio e breve termine 40](#_Toc139975314)

[Scheduling con priorità 41](#_Toc139975315)

[Message Passing (Scambio di Messaggi) 42](#_Toc139975316)

[Send e receive 42](#_Toc139975317)

[Deadlock 43](#_Toc139975318)

[Deadlock nell’allocazione di risorse 43](#_Toc139975319)

[Condizioni per un deadlock di risorsa 44](#_Toc139975320)

[Modelli dello stato di allocazione delle risorse 44](#_Toc139975321)

[Gestione dei deadlock 46](#_Toc139975322)

[Individuazione (rilevamento) e risoluzione dei deadlock 47](#_Toc139975323)

[Prevenzione dei deadlock 50](#_Toc139975324)

[Evitare i deadlock (Algoritmo del banchiere) 52](#_Toc139975325)

[File System 58](#_Toc139975326)

[Directory 58](#_Toc139975327)

[Organizzazione dell’I/O (DMA) 59](#_Toc139975328)

[Free list 61](#_Toc139975329)

[Gestione delle free list 61](#_Toc139975330)

[Scheduling del disco 62](#_Toc139975331)

[Frammentazione della memoria 63](#_Toc139975332)

[Buddy system e allocatori potenza del 2 63](#_Toc139975333)

[Indirizzo logico, indirizzo fisico e traduzione di indirizzi 65](#_Toc139975334)

[Paginazione 66](#_Toc139975335)

[Tabella delle pagine 69](#_Toc139975336)

[Page fault 69](#_Toc139975337)

[Sostituzione delle pagine 70](#_Toc139975338)

[Tempo effettivo di accesso alla memoria (EAT) 72](#_Toc139975339)

[Allocazione della memoria ad un processo e thrashing 72](#_Toc139975340)

[Traduzione degli indirizzi e generazione del page fault (TLB) 73](#_Toc139975341)

[Organizzazione pratica della PT 74](#_Toc139975342)

[Tabella delle pagine invertite (IPT) 75](#_Toc139975343)

[Tabella delle pagine multilivello 75](#_Toc139975344)

[Politiche di sostituzione delle pagine 76](#_Toc139975345)

[Sostituzione ottimale delle pagine 77](#_Toc139975346)

[Sostituzione delle pagine FIFO 77](#_Toc139975347)

[Sostituzione delle pagine LRU (Least Recently Used) 77](#_Toc139975348)

[Proprietà dello stack 78](#_Toc139975349)

[Politiche di sostituzione delle pagine in pratica (algoritmi di sostituzione) 79](#_Toc139975350)

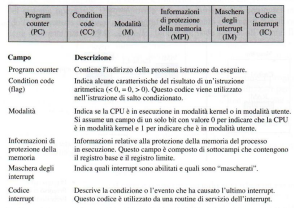
[Working set 80](#_Toc139975351)

[Copy-on-write 82](#_Toc139975352)

## CPU (Program Status Word)

Le componenti della CPU visibili ai programmi utente e al SO sono:

* registri general purpose (GPR), che vengono utilizzati per memorizzare dati, indirizzi, valori o stack pointer durante l’esecuzione di un programma;
* registri di controllo, che contengono informazioni finalizzate al controllo delle operazioni della CPU. L’insieme di questi registri prende il nome di Program Status Word (PSW); ciascun registro è un campo del PSW.



GPR e PSW contengono tutte le informazioni necessarie a conoscere lo stato della CPU. Come visto in precedenza, il kernel può prelazionare il programma attualmente in esecuzione dalla CPU, ne salva lo stato nei GPR e nel PSW, in modo tale da caricare queste informazioni quando il programma verrà ripristinato.

## System call

Un programma, durante la sua esecuzione, potrebbe aver bisogno di utilizzare determinate risorse del computer, come ad esempio le periferiche di I/O. Considerando però che le risorse sono condivise, è necessario prevenire possibili interferenze nel loro utilizzo, per questo le istruzioni che allocano o hanno accesso ad esse, sono istruzioni privilegiate, ciò significa che la CPU può accedervi solo in modalità kernel. Una chiamata di sistema (system call) è una richiesta che un programma fa al kernel attraverso interrupt software.

## Stato di un processo e transizioni di stato

Lo stato di un processo è l’indicatore che descrive la natura dell’attività corrente di un processo. Un processo ha **4 stati fondamentali**:

* **running** (**in esecuzione**), quando la CPU sta eseguendo le istruzioni del processo stesso;
* **blocked** (**bloccato**), quando il processo è in attesa che una risorsa da lui richiesta gli venga allocata, oppure quando non si è ancora verificato uno specifico evento;
* **ready** (**pronto**), quando un processo può essere schedulato ma non è stato ancora eseguito il dispatch;
* **terminated** (**terminato**), quando l’esecuzione del processo è stata completata correttamente, oppure è stata terminata dal kernel.

Un computer che ha una sola CPU può avere un solo processo nello stato di running, ma più processi negli stati blocked, ready e terminated. Un processo può entrare più volte negli stati ready, running e blocked, ma una sola volta nello stato terminated.

Immagine che contiene diagramma

Descrizione generata automaticamente

### Processi sospesi

Per descrivere i processi sospesi per lo swapping, un kernel ha bisogno di stati aggiuntivi. Un processo è nello stato **ready-swapped** quando viene swappato sul disco e rimane in attesa: per riprendere l’esecuzione deve essere riportato in memoria centrale. Un processo è nello stato **blocked-swapped** se l’utente specifica che il processo non deve essere considerato schedulabile per un certo periodo di tempo; anche in questo caso viene swappato sul disco.

Immagine che contiene diagramma

Descrizione generata automaticamente

### Process Control Block

Il Process Control Block contiene lo **stato di un processo** e lo **stato della CPU** relativo ad esso, quando la CPU non sta eseguendo le sue istruzioni. **Campi del Process Control Block** (**PCB**):

* **id del processo**, l'id univoco assegnato al processo al momento della creazione;
* **id del genitore e dei figli**, id utilizzati per la sincronizzazione dei processi, tipicamente per consentire a un processo di verificare se un processo figlio ha terminato la sua esecuzione;
* **priorità**, generalmente un valore numerico. Al momento della creazione, al processo viene assegnata una priorità. Il kernel può cambiare la priorità dinamicamente in base alla natura del processo (**CPU-bound** o **I/O-bound**), al tempo di attività e alle risorse utilizzate (solitamente il tempo di CPU);
* **stato del processo**, appunto lo stato corrente del processo;
* **PSW**, in particolare il contenuto del PSW l'ultima volta che il processo è stato bloccato o prelazionato. Il caricamento di questi valori nel PSW ripristina l'esecuzione del processo;
* **GPR**, cioè il contenuto dei registri general purpose salvati l'ultima volta che il processo è stato bloccato o prelazionato;
* **informazione sugli eventi**, per quanto riguarda un processo nello stato blocked, questo campo contiene l'informazione relativa all'evento per il quale il processo è in attesa;
* **informazioni sui segnali**, in particolare relativa ai gestori dei segnali;
* **puntatore al PCB**, questo campo è utilizzato per creare una lista di PCB, necessaria per la schedulazione da parte del kernel.

## Sincronizzazione dei processi

La sincronizzazione dei processi indica le tecniche utilizzate per ritardare e ripristinare i processi, per implementare le interazioni tra essi. Esistono però anche processi che non interagiscono tra loro; questi prendono il nome di processi indipendenti.

## Race condition

Gli accessi non coordinati ai dati condivisi possono compromettere la consistenza dei dati. Due processi si dicono **interagenti** se l’insieme dei dati modificati da uno dei processi si sovrappone con l’insieme dei dati letti o modificati dell’altro. Se consideriamo due processi eseguiti in modo concorrente (la fine di un processo avviene dopo l’inizio dell’altro), che utilizzano le stesse risorse, potremmo trovarci davanti ad una situazione di race condition. La race condition è una condizione in cui il valore di un oggetto condiviso, risultante dall’esecuzione delle operazioni sui due processi interagenti, può essere diverso. Per prevenire questa condizione, viene garantito che le operazioni dei due processi non vengano eseguite in modo concorrente, e questo avviene mediante **mutua** **esclusione**, ossia solo un’operazione accede ai dati in ogni istante. La **sincronizzazione per l’accesso ai dati** è il coordinamento dei processi per implementare la mutua esclusione su dati condivisi. Per prevenire una race condition si controlla che l’insieme dei dati aggiornati (letti,modificati o scritti) da un processo, non si intersechi con l’insieme dei dati aggiornati da un altro (processo).

## Sezioni critiche (SC)

La mutua esclusione è implementata utilizzando le sezioni critiche (SC) di codice, ossia sezioni del codice che possono essere eseguite da un solo processo per volta. In pratica se un processo sta eseguendo una SC ed un altro processo vuole eseguire quella stessa SC, quest’ultimo dovrà attendere finchè il primo processo non termini l’esecuzione della sua SC. Utilizzare SC causa però ritardi nelle operazioni dei processi, di conseguenza questi non devono essere eseguiti a lungo in una SC. Un processo non deve invocare system call che possono portarlo in uno stato bloccato quando esso si trova all’interno di una SC, ed il kernel, che dovrebbe essere sempre informato se un processo si trova in una SC, non deve prelazionare un processo che ne sta eseguendo una.

### Proprietà di un’implementazione di una sezione critica

**Proprietà** **essenziali** di un’implementazione di una SC:

* **mutua** **esclusione**, in ogni momento solo un processo può accedere alla sezione critica per un particolare oggetto;
* **progresso**, quando nessun processo sta eseguendo una SC per un particolare oggetto, verrà garantito l’accesso ad uno dei processi che desidera eseguirla per utilizzare quello stesso oggetto;
* **attesa** **limitata**, dopo che un processo ha indicato il desiderio di entrare in una sezione critica per un determinato oggetto, il numero dei processi che può guadagnare l’accesso alla SC per questo oggetto, davanti al processo inizialmente nominato, è limitato da un intero finito.

Il progresso e l’attesa limitata prevengono il fenomeno della **starvation**, secondo cui un processo meno prioritario debba attendere un tempo infinito per accedere ad un dato oggetto, poiché continuano ad essergli messi davanti processi a priorità più alta che hanno richiesto l’entrata nella SC successivamente.

## Sincronizzazione di controllo e operazioni indivisibili

I **processi** **interagenti** devono coordinare la loro esecuzione tra loro, per eseguire le rispettive azioni nell’ordine desiderato. Questo requisito viene soddisfatto mediante la sincronizzazione di controllo. La **segnalazione** è una tecnica generale di sincronizzazione di controllo, ma non può funzionare poiché i processi potrebbero bloccarsi indefinitamente in alcune situazioni. Vengono perciò utilizzate le **operazioni** **indivisibili** (o **atomiche**), ossia operazioni che assicurano l’esecuzione di una sequenza di istruzioni, senza che i processi vengano prelazionati. Un’operazione indivisibile, quindi, non può essere eseguita in modo concorrente con un’altra operazione che coinvolge gli stessi dati.

Immagine che contiene testo, lettera

Descrizione generata automaticamente

## Approcci alla sincronizzazione

Esistono **vari approcci alla sincronizzazione**:

* **ciclare vs bloccare**;
* **supporto HW per la sincronizzazione dei processi**;
* **approcci** **algoritmici**, per implementare la mutua esclusione, in modo indipendente dalla piattaforma HW o SW;
* **primitive di sincronizzazione**, che vengono implementate utilizzando operazioni indivisibili, con supporto del kernel, come ad esempio wait e signal dei semafori. Questo tipo di approccio ha un problema legato all’utilizzo alla rinfusa di queste operazioni;
* **costruttori di programmazione concorrente**, come ad esempio i monitor.

### Ciclare vs bloccare

Un processo che vuole eseguire la sua sezione critica deve controllare che la variabile globale condivisa **turno** abbia come valore l’identificatore del suo processo, altrimenti lo stesso processo resta in **busy** **wait** (**attesa** **attiva**), continuando a ciclare, aspettando che la variabile turno assuma valore uguale al suo identificatore. In pratica il processo controlla in un ciclo while se qualche processo è già in sezione critica o sta svolgendo un’operazione indivisibile, utilizzando lo stesso dato, ed in caso negativo entra in sezione critica, terminando l’attesa attiva. Un’attesa attiva ha molte conseguenze negative, infatti non può fornire la proprietà di **attesa** **limitata**, porta a **degrado** **delle** **prestazioni** causate dal continuo ciclare, e può anche causare **deadlock**. Per evitare le attese attive, un processo in attesa di entrare in una SC è posto nello stato blocked, che viene cambiato in ready solo quando può entrare nella SC. Entrambi i metodi possono evitare le race condition, ma lo fanno in modo diverso, in quanto nel caso del ciclare si utilizza un approccio algoritmico, ossia una complessa organizzazione di controlli, mentre nel caso del bloccare si fa uso di caratteristiche HW del computer.

### Supporto HW per la sincronizzazione dei processi

Nell’approccio hardware la sincronizzazione tra processi viene implementata mediante **istruzioni** **indivisibili** (particolari instruzioni macchina), che servono ad evitare race condition sulle locazioni di memoria. Per implementare le operazioni indivisibili e le SC vengono utilizzate queste istruzioni indivisibili, unite alle **variabili** **di** **lock**. Queste variabili hanno due stati: aperto, con valore 0, e chiuso, con valore 1, e per cambiarlo (il valore) vengono utilizzate le istruzioni indivisibili. Il test per poter entrare nella SC viene eseguito anch’esso mediante un’istruzione indivisibile (istruzione test-and-set o istruzione swap).

Immagine che contiene diagramma

Descrizione generata automaticamente

**Vantaggi** dell’approccio HW:

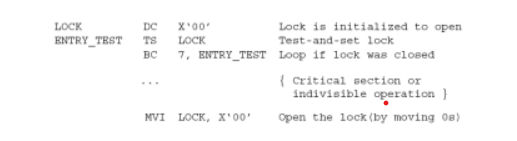
* applicabile su sistemi con un qualsiasi numero di processori;
* semplice e facile da verificare;
* garantisce la mutua esclusione.

**Svantaggi** dell’approccio HW:

* **busy** **waiting**, ossia i processi attendono di eseguire la sezione critica, restando in attesa attiva, quindi la CPU viene utilizzata inutilmente per i cicli di attesa;
* **starvation**, non è garantita l’attesa limitata, in quanto un processo potrebbe non essere mai scelto per entrare in SC.

## Istruzione test-and-set

L’istruzione test-and-set è un’**istruzione indivisibile** che permette l’implementazione di SC o operazioni indivisibili, mediante il seguente algoritmo:



Per utilizzare l’istruzione utilizzeremo una variabile condivisa lock, per coordinare l’accesso alla memoria condivisa. Quando lock è a 0, qualunque processo può metterla ad 1 utilizzando l’istruzione test-and-set, e poi leggere o scrivere nella memoria condivisa. Quando il processo ha finito, reimposta il valore di lock a 0. Esempio di utilizzo:

Immagine che contiene testo, lettera

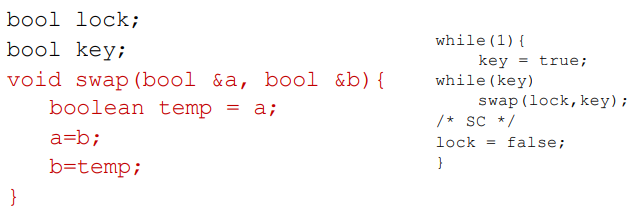
Descrizione generata automaticamente

## Istruzione swap

L'istruzione "swap" è un’istruzione indivisibile la quale garantisce che solo un processo alla volta possa accedere alla risorsa condivisa. Quando un processo esegue l'istruzione "swap", viene passato un valore e un indirizzo di memoria come parametri. L'istruzione "swap" scambia il valore della memoria all'indirizzo specificato con il valore fornito, e restituisce il valore precedente presente nella memoria. In questo modo, se due o più processi eseguono l'istruzione "swap" contemporaneamente, solo uno di essi avrà successo nell'effettuare lo scambio dei valori, mentre gli altri falliranno. Il processo che ha eseguito l'operazione con successo è quindi in grado di accedere alla risorsa condivisa in modo sicuro, senza interferenze da altri processi. Implementazione di SC o operazione indivisibile con istruzione swap:



Esempio di utilizzo dell’istruzione swap:



## Introduzione ai problemi di sincronizzazione classici

Gli elementi chiave per la sincronizzazione dei processi sono le SC e l’utilizzo dei segnali, per cui una soluzione ad un problema di sincronizzazione dovrebbe contenere una combinazione adeguata di questi elementi. Una soluzione, inoltre, dovrebbe soddisfare i criteri di **correttezza**, **massima** **concorrenza** e **nessuna** **attesa** **attiva**.

### Produttori-consumatori con buffer limitato

Questo problema si compone di un **numero non specificato di** **produttori** e **consumatori**, e di un **insieme finito di** **buffer**, ciascuno dei quali è in grado di contenere un elemento di informazione. Un buffer è **pieno** quando un produttore scrive un nuovo elemento al suo interno, mentre è **vuoto** quando un consumatore estrae un elemento contenuto in esso. Un **processo** **produttore** produce un elemento di informazione alla volta, e lo inserisce in un buffer vuoto, mentre un **processo** **consumatore** estrae l’informazione, un elemento alla volta, da un buffer pieno. Una **soluzione** **al** **problema** produttori-consumatori deve soddisfare le seguenti condizioni:

1. un produttore non deve sovrascrivere un buffer pieno;
2. un consumatore non deve consumare un buffer vuoto;
3. i produttori ed i consumatori devono accedere ai buffer in maniera mutuamente esclusiva;
4. (opzionale) le informazioni devono essere consumate nello stesso ordine con il quale sono state inserite nel buffer, secondo la politica FIFO.

**Produttori-consumatori mediante SC**:

Immagine che contiene tavolo

Descrizione generata automaticamente

Il produttore utilizza la variabile booleana prodotto per interrompere il ciclo while dopo aver prodotto un elemento, il consumatore, invece, utilizza la variabile booleana consumato per interrompere il ciclo while dopo che ha consumato un elemento. Nella SC il produttore controlla ripetutamente se esistono buffer vuoti e non appena ne trova uno inserisce l’elemento nel buffer, ed imposta a true la variabile prodotto. Nella SC il consumatore controlla ripetutamente se esistono buffer pieni e non appena ne trova uno estrae l’elemento dal buffer, ed imposta a true la variabile consumato. Questa soluzione ha **due problemi**:

* **poca** **concorrenza**, dovuta al fatto che gli accessi al buffer si trovano in SC e quindi può accedervi (al buffer) un solo processo alla volta, che sia produttore o consumatore, in ogni istante di tempo, anche se abbiamo a disposizione più buffer;
* **attesa** **attiva**, in quanto entrambi i processi vi finiscono (in attesa attiva) quando controllano se ci sono buffer pieni e buffer vuoti.

Per migliorare lo schema precedente si può utilizzare il **meccanismo** **della** **segnalazione**. Dopo che un produttore ha inserito un elemento in un buffer deve segnalarlo al consumatore, mentre dopo che un consumatore ha estratto un elemento dal buffer deve segnalarlo al produttore. Consideriamo ora una soluzione migliorata per **un solo produttore**, **un solo consumatore** ed **buffer limitato**.

Immagine che contiene testo, lettera

Descrizione generata automaticamente**Produttori-consumatori mediante segnalazione**:

Immagine che contiene testo, ricevuta, documento

Descrizione generata automaticamente

Il produttore controlla se il buffer è vuoto attraverso **check\_b\_empty**. Se lo è, produce un elemento, e tramite **post\_b\_full** setta **buffer\_full** a true. Inoltre se il consumatore è blocked, viene sbloccato e attivato. Se invece il buffer è pieno, allora il produttore viene posto a blocked. Per quanto riguarda il consumatore, viene controllato se il buffer è pieno attraverso **check\_b\_full**. Se lo è, estrae un elemento, e tramite **post\_b\_empty** setta buffer\_full a false. Inoltre se il produttore è blocked, viene sbloccato e attivato. Se invece il buffer è vuoto, allora il consumatore viene posto a blocked.

### Lettori e scrittori

Questo algoritmo si compone di un **numero non specificato di lettori e di scrittori**. Un **processo** **letto** può esclusivamente leggere i dati, invece un **processo** **scritto** può modificarli o aggiornarli. Una **soluzione al problema lettori-scrittori** deve soddisfare le seguenti condizioni:

1. molti lettori possono leggere in maniera concorrente;
2. la lettura è proibita mentre uno scrittore sta scrivendo;
3. solo uno scrittore per volta può scrivere;
4. (opzionale) un lettore ha una priorità non prelazionabile sugli scrittori (sistema lettori-scrittori con priorità ai lettori).

Immagine che contiene testo

Descrizione generata automaticamente

Se uno scrittore sta scrivendo, allora il lettore attende, altrimenti il lettore legge, e poi controlla che non ci siano altri lettori che leggono. Se è cosi, e se ci sono scrittori che stanno attendendo, allora ne viene attivato uno. Se invece ci troviamo nel processo scrittore, se ci sono lettori che leggono o uno scrittore che sta scrivendo, allora si attende, altrimenti il processo entra in SC e scrive. Dopo aver scritto, se ci sono lettori o scrittori che attendono, viene attivato uno degli scrittori o tutti i lettori che stanno attendendo.

### Filosofi a cena

Immagine che contiene testo

Descrizione generata automaticamenteCi sono **cinque filosofi seduti attorno ad un tavolo**, ogni filosofo ha davanti a sè un piatto, e **tra ogni coppia di filosofi c’è una forchetta**. Ciascun filosofo (che equivale ad un processo) deve poter mangiare, prendendo una forchetta per volta, quando ha fame, e nessuno dovrebbe morire di inedia (estenuazione dell’organismo dovuta a mancanza di alimenti). Risulta evidente che il numero di forchette impedisce a tutti i filosofi di mangiare contemporaneamente, quindi una corretta programmazione concorrente deve essere in grado di far mangiare alternatamente tutti i filosofi, evitando che qualcuno soffra di **starvation**, ossia un filosofo affamato non attende indefinitamente quando decide di mangiare. Vanno evitati anche i **deadlock**, cioè quando due o più processi rimangono bloccati in attesa che uno esegua una certa azione che serve all’altro e viceversa, e i **livelock**, che si verificano quando due o più processi cambiano continuamente stato, senza che nessuno dei due progredisca.

Questo schema può portare a potenziali deadlock o race condition, che potrebbero essere evitate facendo rilasciare la forchetta di sinistra se la forchetta di destra non è disponibile, ma questa soluzione porterebbe a livelock.

Uno **schema migliorato di processo filosofo** prevede delle SC:

Immagine che contiene testo, lettera

Descrizione generata automaticamente

Si utilizza una variabile booleana **successful** per controllare se il filosofo è riuscito a prendere entrambe le forchette. Il ciclo while si ripeterà fino a quando successful sarà false: nel caso in cui lo sia, si entra in SC e si controlla se sono disponibili entrambe le forchette, le si prende una alla volta, e si setta successful a true. Così facendo, non si possono verificare race condition, in quanto ci troviamo in una SC. Se all’uscita dalla SC successful è ancora false, allora si blocca il processo filosofo, altrimenti il filosofo mangia, ed in una nuova SC posa entrambe le forchette. A questo punto, se il filosofo a sinistra sta aspettando la forchetta destra, allora si attiva il processo riguardante lo stesso filosofo e si fa la stessa cosa per il filosofo destro se sta aspettando la forchetta sinistra. Questa struttura assicura che almeno alcuni filosofi possano mangiare ad ogni istante di tempo, prevenendo situazioni di deadlock. Un filosofo che non riesce a prendere entrambe le forchette allo stesso tempo si blocca, e verrà riattivato quando uno dei suoi vicini posa una forchetta condivisa. Pertanto il processo bloccato deve controllare nuovamente la disponibilità delle forchette, cioè lo scopo del while. Il ciclo, tuttavia, causa una **condizione di attesa attiva**.

### 

## Approcci algoritmici per le SC

Gli approcci algoritmici per implementare le SC non impiegano i servizi del kernel per il blocco e l’attivazione dei processi, utili a ritardare un processo, e le istruzioni indivisibili HW, che servono per evitare le race condition. Quindi sono indipendenti dal SO e dall’HW, ma al loro posto utilizzano il **busy** **waiting** (**attesa** **attiva**) per ritardare un processo, e una **complessa organizzazione di condizioni logiche** per evitare le race condition. Questo tipo di approcci algoritmici per le SC si divide in **algoritmi a due processi** e **algoritmi a n processi**.

### Algoritmi a due processi

**Prima soluzione**:

Immagine che contiene testo, lettera

Descrizione generata automaticamente

La **variabile** **turn** è una variabile condivisa che indica quale sarà il prossimo processo ad accedere alla sezione critica. Essa può assumere valore 1 o valore 2, e viene inizializzata a 1 prima che vengano creati i due processi. Se P1 vuole entrare in SC, bisogna controllare che il valore di turn sia uguale a 1. Se lo è, P1 entra in SC, esegue le operazioni, e all’uscita setta turn a valore 2, in modo che P2 possa successivamente entrare in SC. Se P2 vuole entrare in SC, bisogna controllare che il valore di turn sia uguale a 2. Se lo è, P2 entra in SC, esegue le operazioni, e all’uscita setta turn a valore 1, in modo che P1 possa successivamente entrare in SC. In questo modo è garantita la **mutua** **esclusione**, ma questo algoritmo **soffre di busy waiting**. Il processo infatti non può fare niente fino a quando non avrà il permesso di entrare in SC, che può avvenire solo controllando il valore della variabile turn, con il processo che consuma tempo di CPU, rimanendo attivo mentre aspetta. Come abbiamo detto in precedenza quindi, questa soluzione grantisce la mutua esclusione, ma **porta possibilità di starvation**, in quanto non viene soddisfatta la proprietà dell’attesa limitata, e soprattutto **viola la condizione del progresso**, poichè se un processo fallisce, l’altro rimane bloccato per sempre. **Seconda soluzione**:

Immagine che contiene testo, ricevuta

Descrizione generata automaticamente

Il problema della prima soluzione è che si mantengono informazioni riguardo l’identificatore del processo che può accedere alla SC, ma in realtà sarebbe utile mantenere le informazioni riguardanti lo stato di entrambi i processi, in modo che ogni processo possa accedere alla propria SC anche in caso di fallimento dell’altro. Abbiamo quindi due variabili c1 e c2 che sono rispettivamente **flag di stato** per P1 e P2, che sono poste a 0 quando i processi si trovano nelle rispettive SC e ad 1 quando non si trovano in SC. All’interno del ciclo while inerente a P1 si controlla se P2 si trova in SC. Se si trova in SC, allora si continua a ciclare, altrimenti viene settata c1 a 0, e P1 entra in SC. All’uscita viene settata c1 a 1, in modo che P2 possa poi entrare in SC. All’interno del ciclo while inerente a P2, invece, si controlla se P1 si trova in SC. Se si trova in SC, allora si continua a ciclare, altrimenti viene settata c2 a 0, e P2 entra in SC. All’uscita viene settata c2 a 1, in modo che P1 possa poi entrare in SC. In questo algoritmo **non vi è violazione di progresso**, in quanto se un processo fallisce all’esterno della sua SC (che idealmente comprende anche il codice per modificare il flag), l’altro processo non è bloccato, e può entrare nella propria SC tutte le volte che vuole, visto che il flag dell’altro processo è sempre falso. Se invece un processo fallisce all’interno della propria SC, o dopo aver messo il flag a vero prima di entrare, allora l’altro processo è bloccato per sempre. Questo algoritmo inoltre **non garantisce la mutua esclusione** quando i due processi tentano di accedere contemporaneamente alle rispettive SC, in quanto sia c1 che c2 avranno valore 1, e non essendo ancora nelle SC, entrambi i processi vi entreranno. Si potrebbe perciò pensare di invertire le righe di codice inerenti al while e al cambio di flag, in modo da effettuare il cambio di flag prima del controllo nel while, ma così facendo, se entrambi i processi tentassero di entrare nelle SC contemporaneamente, sia c1 che c2 avrebbero valore 0, per cui entrambi i processi aspetterebbero l’un l’altro indefinitamente, **causando deadlock**.

### Algoritmi a due processi (Algoritmo di Dekker)

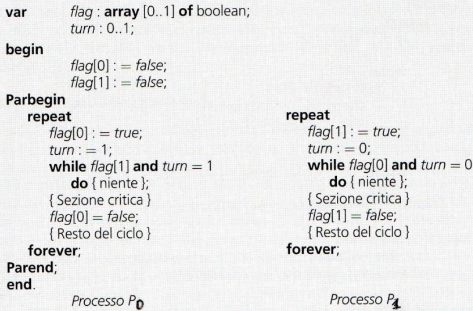
Visto che nessuna delle due soluzioni precedenti risulta sufficiente, l’algoritmo di Dekker le combina.

Immagine che contiene testo, ricevuta

Descrizione generata automaticamente

Analizziamo il processo P1: l’istruzione prima del while serve ad indicare che P1 vuole entrare in SC, quindi mette il proprio flag di stato a 0. Il ciclo while serve a controllare se anche P2 vuole entrare in SC: se è falso, allora P1 può entrare in SC saltando il ciclo while, e all’uscita viene impostata turn a 2, per permettere a P2 di entrare nella sua SC, ed il suo flag di stato c1 ad 1, poichè P1 è uscito dalla sua SC. Se invece P2 vuole entrare anch’esso in SC, viene verificato il valore della variabile turn. Se ha valore 1 deve essere favorito P1, se invece ha valore 2 deve essere favorito P2. Se viene favorito P1 allora viene impostato il flag di stato c1 a 0 in quanto gli è permesso entrare in SC, se invece viene favorito P2, P1 si pone in attesa attiva fino a che P2 non esce dalla sua SC. Analizziamo il processo P2: l’istruzione prima del while serve ad indicare che P2 vuole entrare in SC, quindi mette il proprio flag di stato a 0. Il ciclo while serve a controllare se anche P1 vuole entrare in SC: se è falso, allora P2 può entrare in SC saltando il ciclo while, e all’uscita viene impostata turn a 1, per permettere a P1 di entrare nella sua SC, ed il suo flag di stato c2 ad 1, poichè P2 è uscito dalla sua SC. Se invece P1 vuole entrare anch’esso in SC, viene verificato il valore della variabile turn. Se ha valore 1 deve essere favorito P1, se invece ha valore 2 deve essere favorito P2. Se viene favorito P2 allora viene impostato il flag di stato c2 a 0 in quanto gli è permesso entrare in SC, se invece viene favorito P1, P2 si pone in attesa attiva fino a che P1 non esce dalla sua SC. In pratica se P1 e P2 provano ad entrare contemporaneamente nelle rispettive SC, allora turn indica a quale dei due è consentito, mentre se non c’è competizione per entrare, allora turn non ha effetto. Tramite la **combinazione delle due soluzioni precedenti**, l’algoritmo soddisfa la **mutua esclusione** ed **evita situazioni di deadlock**.

### Algoritmi a due processi (Algoritmo di Peterson)

L’algoritmo di Peterson, così come l’algoritmo di Dekker, risolve il problema della **mutua esclusione**, ma in modo più semplice rispetto all’algoritmo di Dekker. Al posto delle variabili di stato c1 e c2 dell’algoritmo di Dekker, l’algoritmo di Peterson utilizza un **array booleano**, chiamato **flag**, che contiene un flag per ogni processo. La variabile turn, che indica a chi spetta entrare in SC (se vale 0 favorisce il processo P0, mentre se vale 1 favorisce il processo P1), serve ad **evitare i livelock**. Supponiamo di avere due processi, P0 e P1, con gli id, 0 e 1, che vengono utilizzati come indice dell’array booleano flag per accedere ai flag di stato.

Inizialmente vengono posti a false i due flag di stato riguardanti i due processi. Consideriamo il processo P0: se vuole entrare in SC viene impostato il suo flag di stato, ossia l’elemento di indice 0 dell’array flag, a true, dopodichè cerca di favorire l’altro processo, impostando la variabile turn a 1. A questo punto P0 controlla se P1 è in SC, cioè se il suo flag è impostato a true, e se turn ha valore 1. In caso affermativo, P0 continua a ciclare (attesa attiva), in caso contrario entra in SC. Se entrambi i processi vogliono entrare in SC, il valore di turn determina quale processo può entrare nella propria SC. All’uscita dalla SC, P0 imposta il suo flag di stato a false, in modo da permettere all’altro processo di poter accedere in SC. Ragionamento analogo vale nel caso in cui si consideri il funzionamento di P1: se vuole entrare in SC viene impostato il suo flag di stato, ossia l’elemento di indice 1 dell’array flag, a true, dopodichè cerca di favorire l’altro processo, impostando la variabile turn a 0. A questo punto P1 controlla se P0 è in SC, cioè se il suo flag è impostato a true, e se turn ha valore 0. In caso affermativo, P1 continua a ciclare (attesa attiva), in caso contrario entra in SC. Se entrambi i processi voglio entrare in SC, il valore di turn determina quale processo può entrare nella propria SC. All’uscita dalla SC, P1 imposta il suo flag di stato a false, in modo da permettere all’altro processo di poter accedere in SC. Questo algoritmo **garantisce la mutua esclusione**, e non è possibile che i due processi si blocchino a vicenda, in quanto se ad es. P0 è bloccato nel suo ciclo while, allora il flag di stato di P1 è settato a true e turn vale 1. P0 però può entrare in SC solo quando il flag di stato di P1 è settato a false, oppure quando turn vale 0.

## Algoritmi con n processi

Se si utilizza l’approccio algoritmico per implementare una SC, **l’algoritmo deve conoscere il numero di processi che** la **utilizzano** (**la SC**) per gli stessi dati. Questa conoscenza serve per vari motivi: per la **dimensione dell’array dei flag di stato**, **per i controlli** **atti a determinare se altri processi desiderano entrare in SC** e **per il meccanismo** **con cui un processo ne favorisce un altro**. Se si considerano solo **2 processi**, ogni processo deve controllare solo lo **stato di un altro processo**, e nel caso favorirne l’esecuzione, per assicurare la **correttezza** ed **evitare deadlock**. Con un problema a n processi, ogni processo deve controllare lo **stato di altri n-1 processi**, in modo da **prevenire race condition**. Per questa ragione, gli algoritmi per n processi sono più complessi.

### Algoritmi con n processi (Algoritmo del panettiere)

L’idea dell’algoritmo del panettiere consiste nel far prendere ad ogni processo un numero, e di **servire quello con numero più piccolo**; con **servire** si intende far entrare un processo in SC. In questo algoritmo si utilizzano **due array**:

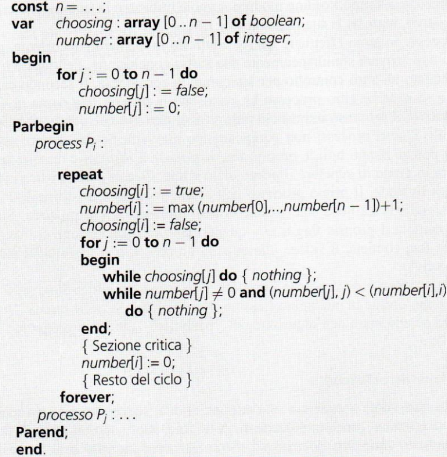
* **choosing**, che è un array di flag booleano che indica se il processo i-simo è impegnato in quell’istante nella scelta di un numero;
* **number**, che al passo i-simo contiene il numero scelto da . E’ importante ricordare che se number al passo i-simo vale 0, significa che non ha scelto un numero dall’ultima volta in cui è entrato in SC.

Poichè l’algoritmo del panettiere non può assicurare che due processi non ricevano lo stesso numero, a parità di numero viene servito il processo con nome minore.

Immagine che contiene testo, Carattere, schermata, Blu elettrico

Descrizione generata automaticamente

**Funzionamento**:



Prima dell’inizio di un qualsiasi processo, tutti i flag booleani dei processi vengono messi a false, in quanto ancora nessun processo è impegnato nella scelta di un umero, e tutti i numeri dei processi sono messi a 0, in quanto non hanno ancora scelto un numero. All’entrata in un qualsiasi processo viene settato il flag corrispondente nell’array choosing, ossia choosing[i], e viene scelto un numero per il processo, che sarà number[i], assegnato come massimo tra i numeri di tutti i processi più 1. Avendo scelto un numero per il processo i-simo, il flag di stato nell’array choosing viene posto a false, e parte il ciclo for riguardante tutti gli altri processi. All’interno del ciclo for troviamo due cicli while: il primo controlla se i processi j-simi (quindi che vanno da 0 a n-1 nel ciclo for) hanno il flag di choosing settato a true, poichè in quel caso si deve attendere che scelgano un numero, mentre il secondo controlla se ci sono processi che, avendo già scelto un numero, hanno la coppia (number[j],j) < (number[i],i). In questo modo, un processo che vuole entrare in SC favorisce un processo con un numero minore, o anche con un numero uguale ma con id minore. Alla fine del ciclo for il processo i-simo può entrare in SC; ed alla sua uscita viene impostato il suo numero a 0, in quanto dovrà scegliere un nuovo numero ogni volta che vuole entrare in SC.

**Considerazioni**: se non si utilizzasse l'array choosing allora il processo che ha appena eseguito la SC potrebbe tentare immediatamente di entrarci di nuovo, senza dare la possibilità ad altri processi di eseguire la loro SC. Ciò potrebbe causare un **possibile** **problema di starvation** (alcuni processi potrebbero essere bloccati e non essere in grado di eseguire la loro SC). L'array choosing serve quindi a garantire un ordinamento tra i processi che richiedono l'accesso alla SC. In questo modo, i processi vengono scelti in ordine di priorità e viene evitato il problema di starvation.

## Semafori

Un semaforo è una particolare struttura di sincronizzazione che consiste in una **variabile intera condivisa**, con valori non negativi, che può essere soggetta solo alle **operazioni di inizializzazione**, specificate come parte della dichiarazione, e alle **operazioni indivisibili wait e signal** (o post).

Immagine che contiene testo, lettera

Descrizione generata automaticamente

Quando un processo effettua un’operazione **wait** su un semaforo, l’operazione verifica se il valore del semaforo è maggiore di 0. In caso affermativo, decrementa il valore del semaforo di 1 e consente al processo di proseguire la sua esecuzione, altrimenti blocca il processo sul semaforo. Un’operazione **signal** su un semaforo, invece, attiva un processo bloccato sul semaforo, se ce ne sono, oppure incrementa il valore del semaforo di 1. A causa di queste semantiche, i semafori sono anche chiamati **semafori** **contatori**. L’indivisibilità delle operazioni wait e signal è assicurata dal linguaggio di programmazione o dal SO che le implementa, assicurando che non si possano verificare race condition su un semaforo.

### Utilizzo dei semafori nei sistemi concorrenti

**Tre utilizzi dei semafori** nell’implementazione dei sistemi concorrenti sono:

* la **mutua esclusione**, utile per implementare le SC, la concorrenza limitata e la segnalazione tra processi. Es. :

Immagine che contiene testo, ricevuta, Carattere, algebra

Descrizione generata automaticamente

Consideriamo sem\_CS un semaforo inizializzato al valore 1 e e due processi. Ogni processo effettua un’operazione wait su sem\_CS, prima di entrare nella propria SC, e un’operazione signal, all’uscita dalla sua SC. Il primo processo ad effettuare la wait trova che sem\_CS è maggiore di 0, quindi decrementa sem\_CS di 1 e procede entrando nella sua SC. Quando il secondo processo effettua la wait sul semaforo, questo si blocca poichè sem\_CS vale 0. Viene successivamente riattivato quando il primo processo effettua l’operazione di signal su sem\_CS, ossia all’uscita dalla sua SC. Se non ci sono processi bloccati su sem\_CS quando viene effettuata la chiamata a signal, allora il valore di sem\_CS viene incrementato di 1. Questo valore di sem\_CS (1) consente ad un processo di entrare immediatamente nella SC, ovviamente dopo aver effettuato un’operazione di wait sul semaforo;

* la **concorrenza** **limitata**, importante quando una risorsa può essere condivisa da al più c processi, che possono effettuare in maniera concorrente un operazione , con c una costante maggiore o uguale di 1. La concorrenza limitata viene implementata inizializzando un semaforo sem\_c a c. Ogni processo che intende eseguire effettua una wait prima di effettuare l’operazione, ed una signal ad operazione conclusa. Dalla semantica di wait e di signal risulta chiaro che al più c processi possono eseguire concorrentemente ;
* la **segnalazione** **tra** **processi**, utile nel controllo della sincronizzazione. Es. :

Immagine che contiene testo

Descrizione generata automaticamente

Consideriamo sync un semaforo inizializzato al valore 0 e e due processi. Qui il processo esegue una wait su sync prima di effettuare un’operazione , e effettua una signal, sempre su sync, dopo aver eseguito l’operazione . Visto che il semaforo sync è inizializzato a 0, il processo verrà bloccato su wait se non ha già effettuato una signal. Non si possono verificare race condition poichè le operazioni di wait e di signal sono indivisibili. L’organizzazione per la segnalazione può essere utilizzata ripetutamente, visto che l’operazione wait riporta il valore di sync a 0.

### Semafori binari

Un semaforo binario è un tipo speciale di semaforo utilizzato per implementare la **mutua** **esclusione**. Può essere utilizzato in maniera intercambiabile con i mutex, oggetti simili ma che hanno come operazioni possibili lock e unlock. Un semaforo binario può assumere solo i valori 0 e 1. Le operazioni wait e signal su un semaforo binario sono leggermente differerenti rispetto a quelle dei semafori classici, infatti l’istruzione S = S – 1 nell’operazione wait è sostituita da S = 0, mentre l’istruzione S = S + 1 nell’operazione signal è sostituita da S = 1. La differenza tra un semaforo e un semaforo binario è solo a livello concettuale, e serve a semplificare il lavoro del programmatore.

Immagine che contiene testo

Descrizione generata automaticamente

### Produttori-consumatori a buffer singolo con semafori

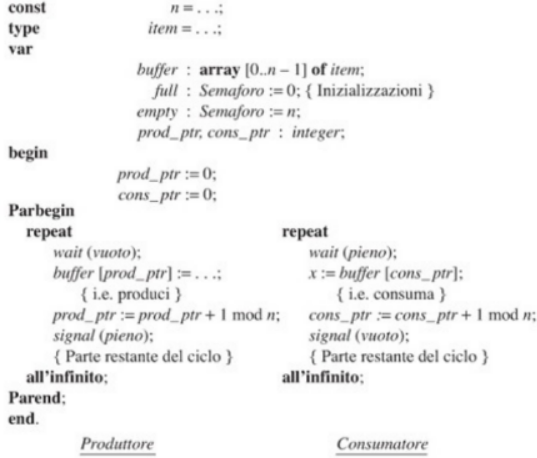
Come già discusso in precedenza, il problema dei produttori-consumatori è un problema di segnalazione. Dopo aver inserito un elemento di informazione in un buffer, un produttore segnala l’evento ad un consumatore in attesa di utilizzare un elemento dal buffer. Bisogna pertanto implementare i produttori-consumatori utilizzando l’organizzazione per la segnalazione tra processi, precedentemente illustrata. In questo caso andiamo a visionare la soluzione per il caso con singolo buffer. Il pool di buffer è rappresentato da un **array di buffer** con un singolo elemento all’interno. Vengono dichiarati **due semafori**, **full** ed **empty**, inizializzati rispettivamente a 0 e 1, utilizzati per indicare se il buffer è pieno o vuoto.

Immagine che contiene testo, ricevuta, schermata

Descrizione generata automaticamente

Inizialmente, il semaforo full ha valore 0, quindi il consumatore si bloccherà sulla wait(full), mentre il semaforo empty ha valore 1, quindi un produttore effettuerà l’operazione di wait(empty), produrrà un elemento, ed infine eseguirà una signal per incrementare il valore di full. Così facendo, permette ad un consumatore di entrare nella propria SC. Quando il consumatore termina l’estrazione (la consumazione dell’elemento) dal buffer, questo (il consumatore) effettua una signal(empty) che consente ad un produttore di effettuare un’operazione di inserimento (produzione). Questa soluzione **evita le attese attive** poichè i semafori sono utilizzati per controllare il buffer full o empty, per cui un processo sarà bloccato se non trova il buffer full o empty come richiesto. Il **livello di concorrenza** in questo sistema è 1, in quanto a volte viene eseguito un produttore e a volte viene eseguito un consumatore.

### Produttori-consumatori a n buffer con semafori

Una semplice estensione della soluzione al problema con singolo buffer, porta all’implementazione della soluzione al problema produttori-consumatori a n buffer, con n maggiore o uguale ad 1.

Anche in questo caso avremo i buffer full ed empty che indicano il numero di buffer rispettivamente pieni e vuoti, ma stavolta saranno inizializzati a 0 e N, invece di 0 e 1. Avremo delle nuove variabili **prod\_ptr** e **cons\_ptr,** che fungeranno da indici dell’array buffer, che verranno rispettivamente incrementate ad ogni inserimento di un elemento nell’array buffer (prod\_ptr) e ad ogni estrazione dallo stesso array (cons\_ptr). Questa caratteristica assicura che i buffer vengano utilizzati in **ordine FIFO**. Un produttore ed un consumatore possono funzionare in modo concorrente finchè vi sono buffer full ed empty nel sistema. Se molti processi produttori e consumatori sono attivi nel sistema, è necessario garantire la **mutua esclusione** sia tra i produttori che tra i consumatori, per **evitare race condition** rispettivamente su prod\_ptr e cons\_ptr.

### Lettori-scrittori con soluzione rifinita mediante contatori

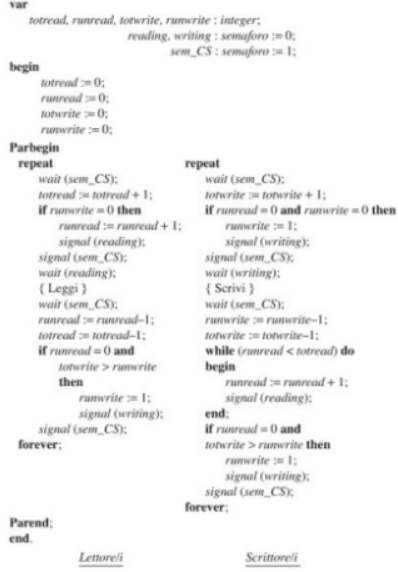
Una caratteristica essenziale del problema dei lettori-scrittori è che i lettori e gli scrittori devono attendere mentre uno scrittore sta scrivendo e, quando lo scrittore termina, o vengono attivati tutti i lettori in attesa o viene attivato uno scrittore in attesa. Per implementare questa caratteristica vengono utilizzati **quattro** **contatori**:

* **runread**, numero di lettori in lettura;
* **totread**, numero di lettori in attesa di lettura o attualmente in lettura;
* **runwrite**, numero di scrittori in scrittura;
* Immagine che contiene testo, schermata, ricevuta, Carattere

  Descrizione generata automaticamente**totwrite**, numero di scrittori in attesa di scrittura o attualmente in scrittura.

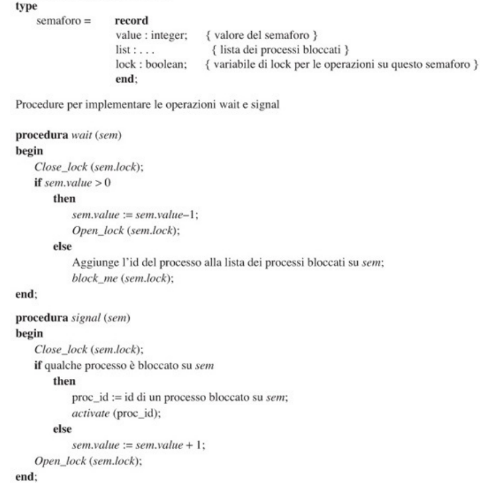
Consideriamo un lettore, questo deve attendere se runwrite è diversa da 0, ossia se ci sono scrittori in scrittura, altrimenti può leggere. Dopodichè, se runread è uguale a 0, ossia se non ci sono lettori in lettura, e totwrite è diversa da 0, ossia se ci sono scrittori in attesa, allora ne viene attivato uno (scrittore in attesa). Per quanto riguarda uno scrittore invece, questo deve attendere se runread o runwrite sono diverse da 0, ossia quando ci sono lettori in lettura o scrittori in scrittura, altrimenti scrive. Dopodichè se totread o totwrite sono diverse da 0, quindi ci sono lettori o scrittori in attesa, viene attivato uno scrittore in attesa o tutti i lettori in attesa. Questa soluzione **non utilizza una SC esplicita per gli scrittori**, che anzi sono bloccati finchè non gli è consentito iniziare a scrivere.

### Lettori-scrittori con semafori (preferenza ai lettori)

Il blocco dei lettori e degli scrittori può essere gestito come nel problema dei produttori-consumatori, ossia utilizzando dei **semafori per la segnalazione**. Vengono quindi introdotti **tre** **semafori**: **reading** e **writing**, che vengono inizializzati a 0, e **sem\_CS**, che viene inizializzato ad 1. Abbiamo sempre runread, totread, runwrite e totwrite, tutte inizializzate a 0, e visto che si utilizzano dei semafori, ovviamente saranno utilizzate le operazioni indivisibili wait e signal.

Un processo lettore esegue una wait(reading) prima di iniziare a leggere, in modo da bloccare il processo se non sono soddisfatte le condizioni di lettura, mentre se sono soddisfatte, il lettore potrà leggere. Analogamente, anche un processo scrittore esegue una wait(writing) prima di iniziare a scrivere, e si blocca se non sono soddisfatte le condizioni appropriate, altrimenti scrive. Le condizioni su cui si bloccano i lettori e gli scrittori possono cambiare quando cambia uno dei contatori, ossia quando un lettore termina la lettura o uno scrittore termina la scrittura. Per questo motivo i processi lettore e scrittore eseguono delle opportune operazioni di signal dopo aver completato un’operazione di lettura o di scrittura. Per evitare race condition, inoltre, tutti i contatori vengono esaminati e modificati all’interno di SC implementate mediante un semaforo binario chiamato sem\_CS. Quando un lettore vuole iniziare una lettura, entra nella SC protetta da sem\_CS per controllare se runwrite è uguale a 0, ossia se non ci sono lettori in lettura. In caso affermativo, incrementa runread, ossia incrementa il numero di lettori in lettura, esce dalla SC e inizia a leggere. In caso negativo invece, deve eseguire una wait(reading), ma visto che eseguire un’operazione del genere all’interno della SC protetta da sem\_CS può causare un **deadlock**, viene eseguita una signal(reading) dopo essere uscito dalla SC. Questo perchè se non venisse effettuata la signal(reading), l’operazione di wait(reading) metterebbe in attesa il lettore che in realtà può leggere. Uno scrittore, in modo analogo, effettua una signal(writing) all’interno della propria SC protetta da sem\_CS ed una wait(writing) al termine della stessa SC.

### Implementazione dei semafori

Per implementare i semafori viene definito appunto un **tipo semaforo**, in cui sono presenti come **campi**: il **valore** del semaforo, una **lista** utilizzata per memorizzare gli id dei processi bloccati sul semaforo ed una variabile di **lock** per garantire l’indivisibilità delle operazioni wait e signal sul semaforo. Queste operazioni sono implementate come procedure che ricevono come parametro una variabile di tipo semaforo.

Per **evitare race condition** durante l’accesso al valore del semaforo, le procedure wait e signal invocano prima la funzione **Close\_lock** per impostare la variabile di lock **sem\_lock**. Close\_lock si serve di un’istruzione indivisibile e di attesa attiva, che però è breve poichè le operazioni wait e signal sono brevi. Entrambe le procedure inoltre invocano anche la funzione **Open\_lock** per reimpostare il lock al termine della loro esecuzione. La procedura wait controlla se il valore del semaforo è maggiore di 0: in caso affermativo lo decrementa di 1 (il valore del semaforo) e invoca Open\_lock, in caso negativo invece, aggiunge l’id del processo alla lista dei processi bloccati sul semaforo ed esegue la chiamata a **block\_me** con la variabile di lock sem\_lock come parametro. Questa chiamata blocca il processo che ha effettuato la chiamata wait ed inoltre rilascia il lock passatogli come parametro. La procedura wait non avrebbe potuto effettuare queste azioni da sola, in quanto se rilasciasse il lock prima di effettuare una chiamata a block\_me si potrebbero verificare delle **race condition**, mentre si creerebbe una situazione di **deadlock** se richiamasse block\_me prima di rilasciare il lock. L’operazione wait ha un **livello di fault molto basso** in diversi sistemi che utilizzano i semafori, ossia i processi che effettuano le operazioni di wait raramente si bloccano. La procedura signal invece controlla se qualche processo è bloccato sul semaforo: in caso affermativo seleziona uno di questi processi e lo riattiva eseguendo la system call **activate**, in caso negativo invece, incrementa il valore del semaforo di 1. Le operazioni di wait e di signal vengono implementate in **tre modi** diversi:

* **a livello kernel**, dove tutti i processi in un sistema possono condividere un semaforo a livello kernel. Tuttavia ogni operazione di wait e di signal corrisponde ad una system call e questo genera un **alto overhead** per l’utilizzo dei semafori;
* **a livello utente**, in cui le operazioni wait e signal sono codificate come procedure di libreria collegate ad un’applicazione, i cui processi possono condividere i semafori a livello utente. Le chiamate block\_me e activate sono di fatto chiamate a funzioni di libreria, che gestiscono in proprio il blocco e la riattivazione dei processi, ed effettuano system call solo quando necessitano dell’assistenza del kernel;
* **in modo ibrido**, anche in questo caso wait e signal sono codificate come procedure di libreria ed i processi di un’applicazione possono condividere i semafori ibridi. Block\_me e activate sono system call fornite dal kernel, quindi verrebbero effettuate da wait e signal solo quando i processi vengono bloccati o riattivati. Come conseguenza del livello di fault molto basso dell’operazione wait, queste system call verrebbero effettuate raramente, per cui un’implementazione ibrida dei semafori genererebbe un **overhead inferiore** rispetto a quella a livello kernel.

## Monitor

Un monitor è un **costrutto di sincronizzazione** che può essere utilizzato da due o più processi per rendere mutuamente esclusivo l’accesso a risorse condivise. Un monitor è una **raccolta di dati, strutture dati e procedure** raggruppate in un modulo. I processi possono invocare le **procedure** di un monitor in ogni momento, in modo da accedere ai dati dello stesso monitor, a cui altrimenti non potrebbero accedere. Un monitor definisce un lock (implicito) e zero o più **variabili di condizione**, per gestire l’accesso ai dati in modo concorrente. Viene utilizzato un lock sia per assicurare che sia attivo un solo processo sul monitor in ogni istante, che per fornire mutua esclusione sui dati condivisi. Le variabili di condizione, invece, permettono ai processi di bloccarsi nella SC, rilasciando il loro lock, e allo stesso tempo bloccando il processo.

Le **operazioni effettuate dal monitor** sono:

* incapsulare i dati condivisi da proteggere;
* acquisire il lock all’inizio;
* operare sui dati condivisi;
* rilasciare il lock temporaneamente se non può completare;
* riacquisire il lock appena può continuare;
* rilasciare il lock alla fine.

### Variabili di condizione

La sincronizzazione nei monitor è supportata dall’utilizzo delle variabili di condizione, contenute ed accessibili solo nel monitor. Le variabili di condizione sono uno speciale tipo di dato, sul quale è possibile effettuare **due operazioni**:

* **cond\_wait**(c), che sospende l’esecuzione del processo che la invoca sulla variabile di condizione c;
* **cond\_signal**(c), che invece riprende l’esecuzione di qualche processo bloccato precedentemente da una cond\_wait sulla stessa variabile di condizione.

Le operazioni cond\_wait() e cond\_signal() sono **diverse rispetto a quelle nei semafori**: l’operazione wait su un semaforo è sospensiva solo se il semaforo non è positivo, mentre nei monitor è immediatamente sospensiva, e l’operazione signal, nei monitor, perde il segnale se, quando un processo la invoca, non c’è nessuno in attesa sulla variabile di condizione. Inoltre la wait e la signal del monitor **non sono contatori**. Dopo una cond\_signal è necessario avere un solo processo attivo nel monitor, di conseguenza abbiamo bisogno di una regola per decidere quale processo bloccato da una cond\_wait precedente risvegliare. Secondo **Hoare** si esegue il processo appena risvegliato, sospendendo quello che ha invocato cond\_signal. Secondo **Brinch Hansen** invece, il processo che invoca cond\_signal deve uscire immediatamente, con l’operazione (cond\_signal) che può apparire solo come istruzione finale di una procedura del monitor. Dopo la cond\_signal, lo **scheduler** seleziona solo uno dei processi in attesa. Questa regola è più semplice sia concettualmente che da implementare. Un’altra regola possibile è quella **alternativa** (**Mesa**), dove il processo P che invoca cond\_signal continua, mentre il processo in attesa comincia dopo che P esce dal monitor.

## Problema del barbiere addormentato

In un negozio di barbiere abbiamo:

* un barbiere;
* N sedie per i clienti in attesa;
* 1 poltrona da barbiere.

**Specifiche**:

* se non ci sono clienti, il barbiere si addormenta sulla poltrona;
* quando arriva un cliente, questo sveglia il barbiere e si fa tagliare i capelli sulla poltrona;
* se arriva un cliente mentre il barbiere sta tagliando i capelli ad un altro cliente, il nuovo arrivato attende su una delle sedie libere;
* se tutte le sedie sono occupate, il cliente se ne va.

Immagine che contiene tavolo

Descrizione generata automaticamenteDichiariamo e inizializziamo ad 1 un semaforo **sem\_CS**, che serve a garantire la mutua esclusione, e a 0 due semafori, **clienteDisponibile** e **poltrona**. Abbiamo altri due dati, **inAttesa**, che è una variabile intera riguardante il numero di clienti in attesa che viene inizializzata a 0, e **SEDIE**, che è una costante con valore N.

Nel processo riguardante il barbiere, abbiamo una wait relativa a clienteDisponibile, ossia fin quando clienteDisponibile varrà 0 significa che non ci sono clientiDisponibili. Se ci sono significa che clienteDisponibile varrà 1, e di conseguenza nella wait avviene il decremento e si va avanti. Successivamente abbiamo un’altra wait, relativa a sem\_CS, che segue lo stesso ragionamento precedente, ossia se il semaforo vale 1 viene decrementato il suo valore e si va avanti, altrimenti il processo viene bloccato. Se si supera anche questa wait, viene decrementato il valore della variabile intera inAttesa e si effettua una signal a poltrona, quindi se c’è qualche processo bloccato su questo semaforo viene attivato, altrimenti si incrementa il valore del semaforo. Stesso discorso vale per la signal successiva su sem\_CS; se viene superata anch’essa, si procede al taglio dei capelli. Per quanto riguarda un processo cliente invece, abbiamo un’operazione di wait su sem\_CS. Se viene superata, si controlla che il numero di clienti inAttesa sia minore del numero massimo di SEDIE. Se le sedie sono tutte piene, allora si effettua una signal di sem\_CS ed il cliente lascia il negozio. Se invece ci sono sedie libere dove sedersi, inAttesa viene incrementata e si effettua una signal relativa a clienteDisponibile. Se non ci sono processi bloccati sul semaforo, si effettua un’ulteriore signal, stavolta riguardante sem\_CS. Il ragionamento anche in questo caso è lo stesso, se non ci sono processi bloccati sul semaforo si effettua una wait su poltrona, ossia si aspetta che la poltrona si liberi. Una volta liberata la poltrona, viene effettuato il taglio dei capelli.

## Scheduling

Una politica di scheduling decide a quale processo debba essere assegnata la CPU in un certo momento. Per questo motivo le decisioni dello scheduler influenzano sia il servizio utente che le prestazioni del sistema. Lo scheduler considera una lista di richieste in attesa di essere elaborate e ne seleziona una per l’elaborazione. Con richiesta si intende l’esecuzione di un job o di un processo.

Immagine che contiene diagramma

Descrizione generata automaticamente

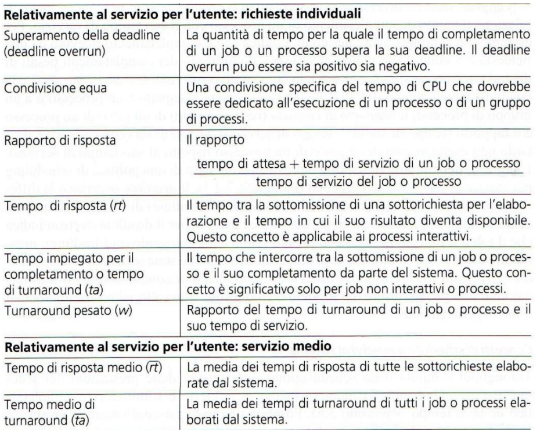
### Terminologia e concetti

Concetti di scheduling **relativi alla richiesta**:

Immagine che contiene tavolo

Descrizione generata automaticamente.

Concetti di scheduling **relativi al servizio per l’utente**:

.

Concetti di scheduling **relativi alle prestazioni**:

Immagine che contiene testo

Descrizione generata automaticamente

### Tecniche di scheduling

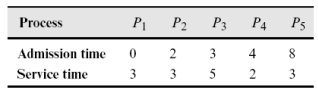
Gli scheduler usano **tre tecniche fondamentali** per ottenere un buon servizio utente ed elevate prestazioni di sistema:

* **scheduling basato su priorità**, ossia il processo in esecuzione è quello con **priorità più alta**. Questo è assicurato selezionando il processo in stato **ready** a più alta priorità e prelazionando il processo in esecuzione, quando un processo con priorità più alta diventa ready. Solitamente si assegna una **priorità alta ai processi I/O bound**, in modo da avere un elevato **throughput** **di** **sistema**;
* **riordino delle richieste**, che implica il fatto che le richieste vengano espletate in ordine diverso rispetto a quello di arrivo. Il riordino è **implicito nella prelazione**, che può essere utilizzata per **migliorare** sia **il servizio utente** (in un sistema time-sharing) che il **throughput di sistema** (in un sistema multiprogrammato);
* **variazione della time slice**, infatti quando si fa uso di questa tecnica si ottengono **tempi di risposta migliori** quando vengono utilizzati valori inferiori per la time slice. In questo modo, però, viene **ridotta** anche **l’efficienza** **della** **CPU**, a causa dei molteplici **context** **switch**.

### Il ruolo delle priorità

La priorità viene utilizzata dallo scheduler quando più processi sono in attesa dell’uso della CPU. La priorità di una richiesta può essere determinata in funzione di vari parametri: si parla di **priorità** **dinamica** se alcuni dei suoi parametri cambiano durante l’elaborazione, altrimenti si parla di **priorità** **statica**. Lo scheduling basato su priorità ha lo svantaggio che una richiesta a bassa priorità potrebbe non essere espletata fintanto che arrivano richieste con priorità più alta, ossia potrebbe portare alla **starvation**. Questa situazione potrebbe essere evitata tramite la tecnica di **aging** **delle** **richieste**, nel quale si incrementa la priorità di una richiesta che non è stata schedulata per un certo lasso di tempo. Se due processi hanno la stessa priorità, invece, si utilizza uno **scheduling round robin**.

## Scheduling senza prelazione

Nello scheduling senza prelazione la CPU serve una richiesta schedulata fino al completamento; di conseguenza lo scheduler ha **soltanto** la funzione di **riordinare le richieste** per migliorare il servizio utente o le prestazioni del sistema. Alcune politiche di scheduling senza prelazione sono **First-Come, First-Served** (**FCFS**), **Shortest Job First** (**SJF**) e **High Response Ratio Next** (**HRN**). Andremo ora a vedere come funzionano queste politiche di scheduling, a partire dai processi

### Scheduling First-Come, First-Served (FCFS)

In questo tipo di scheduling, i processi vengono schedulati nell’**ordine in cui giungono al sistema**, ossia il primo processo ad essere eseguito è quello che per primo ha richiesto la CPU. In pratica i processi ready sono organizzati come una **coda** FIFO e i processi che richiedono la CPU vengono inseriti alla fine di questa coda. Questo tipo di algoritmo è semplice da implementare ma solitamente poco efficiente, considerando il tempo medio d’attesa. E’ un algoritmo che **privilegia** **i** **processi CPU bound**, il che è un problema perchè i processi I/O bound, che richiedono meno tempo di CPU, devono attendere molto tempo di CPU in attesa che finiscano i processi CPU bound, che richiedono appunto molto tempo di CPU.

Immagine che contiene grafico

Descrizione generata automaticamente

Come si può vedere, l’ordine di arrivo dei processi è , che vengono eseguiti proprio nell’ordine in cui sono arrivati. Il processo arriva al tempo 0 e viene subito schedulato in quanto è il primo e unico processo presente nella coda FIFO. Rimane in esecuzione per 3 unità di tempo, per poi rilasciare la CPU. Nel frattempo sono arrivati e , rispettivamente al tempo 2 e al tempo 3, di conseguenza viene schedulato dopo aver atteso per un’unità di tempo il completamento di . Al tempo 3 inizia quindi la sua esecuzione, che termina dopo 3 unità di tempo, cioè al tempo 6. Il suo tempo di turnaround è pari a 4 unità di tempo, in quanto è la somma tra il suo tempo di attesa (1) e il suo tempo di servizio (3). Nel frattempo, al tempo 4, si è aggiunto nella coda il processo . Al completamento di viene schedulato , che era rimasto in attesa per 3 unità di tempo. termina la sua esecuzione dopo 5 unità di tempo, ossia al tempo 11, con un tempo di turnaround pari a 8 unità di tempo (3 di attesa + 5 di servizio). Nel frattempo, al tempo 8, si è aggiunto nella coda anche il processo . Al completamento di viene schedulato , che era rimasto in attesa per 7 unità di tempo. termina la sua esecuzione dopo 2 unità di tempo, ossia al tempo 13, con un tempo di turnaround pari a 9 unità di tempo (7 di attesa + 2 di servizio). Al completamento di , nella coda, vi è solo il processo , che viene schedulato e termina dopo 3 unità di tempo, ossia al tempo 16, con tempo di turnaround pari a 8 unità di tempo (5 di attesa + 3 di servizio).

### Scheduling Shortest Job First (SJF)

Questo tipo di scheduling seleziona il processo in attesa che **utilizzerà la CPU per minor tempo**. Se due processi hanno lo **stesso tempo di esecuzione**, allora verrà applicato lo **scheduling FCFS** precedentemente visto. Questo tipo di algoritmo **eleva il throughput**, ossia il numero di processi portati a termine in un determinato tempo, e **privilegia i processi I/O bound**, poichè sono quelli che utilizzano la CPU per minor tempo. L’algoritmo SJF presenta delle **problematiche**, infatti bisogna conoscere a priori i tempi di servizio dei vari processi e può portare ad un potenziale problema di starvation. Per quanto riguarda la prima problematica, visto che il SO non conosce i tempi di servizio in anticipo, **bisogna effettuare una** **stima delle CPU burst**, ossia le fasi in cui viene impiegata soltanto la CPU senza I/O. Per quanto riguarda il problema di **starvation**, è possibile che un processo (CPU bound solitamente) rimanga in attesa troppo tempo prima di essere completato se vengono aggiunti continuamente piccoli processi alla coda dei processi ready.

Immagine che contiene grafico

Descrizione generata automaticamente

Come abbiamo detto in precedenza, vengono eseguiti i processi in ordine di tempo di servizio minore. Inoltre in questo sistema si assume che un processo ammesso nello stesso istante in cui ne viene sospeso un altro sia inserito nella coda prima del processo sospeso. Al tempo 0 arriva il processo e viene subito schedulato in quanto è l’unico processo presente nella coda. Rimane in esecuzione per 3 unità di tempo, per poi rilasciare la CPU. Nel frattempo sono arrivati nella coda dei processi e , rispettivamente al tempo 2 e al tempo 3, con che viene schedulato poichè ha tempo di servizio minore (3 < 5). va quindi in esecuzione al tempo 3, con la stessa che termina dopo 3 unità di tempo, ossia al tempo 6, con tempo di turnaround pari a 4 unità di tempo (1 di attesa + 3 di servizio). Nel frattempo, al tempo 4, si è aggiunto alla coda anche il processo , che viene schedulato al termine di poichè ha tempo di servizio minore rispetto a (2 < 5). va quindi in esecuzione al tempo 6, con la stessa che termina dopo 2 unità di tempo, ossia al tempo 8, con tempo di turnaround pari a 4 unità di tempo (2 di attesa + 2 di servizio). Nel frattempo, al tempo 8, si è aggiunto alla coda anche il processo , che viene schedulato al termine di poiché ha tempo di servizio minore rispetto a (3 < 5). va quindi in esecuzione al tempo 8, con la stessa che termina dopo 3 unità di tempo, ossia al tempo 11, con tempo di turnaround pari a 3 unità di tempo (0 di attesa + 3 di servizio). Al termine di nella coda rimane solo il processo che quindi viene schedulato. va quindi in esecuzione al tempo 11, con la stessa che termina dopo 5 unità di tempo, ossia al tempo 16, con tempo di turnaround pari a 13 unità di tempo (8 di attesa + 5 di servizio).

### Scheduling High Response Ratio Next (HRN)

Questo algoritmo viene utilizzato per **prevenire l’aging**, ossia l’attesa eccessiva dei processi lunghi che vengono scavalcati da quelli più brevi, cosa che avviene negli algoritmi SJF. Questa politica **calcola i** **tassi di risposta** di tutti i processi nel sistema, secondo la **formula**

**schedulando il processo con rapporto di risposta maggiore**. L’obiettivo è quello di dare la possibilità ai processi con un CPU burst elevato di essere schedulati prima dei processi con CPU burst più piccolo. Il tasso di risposta di un processo breve si incrementa più rapidamente di quello di un processo lungo, ma con il passare del tempo quest’ultimo può diventare sufficientemente grande da consentire al processo di essere schedulato. Con questa tecnica, come detto, si **risolve il possibile problema di starvation** degli algoritmi SJF.

Immagine che contiene testo, diagramma, schermata, numero

Descrizione generata automaticamente

Come detto in precedenza, per la schedulazione occorre prendere in considerazione il tasso di risposta (response ratio) dei processi. Al tempo 0 arriva il processo e viene subito schedulato in quanto è l’unico processo presente nella coda. Rimane in esecuzione per 3 unità di tempo, per poi rilasciare la CPU. Nel frattempo sono arrivati nella coda dei processi e , rispettivamente al tempo 2 e al tempo 3, con che viene schedulato poichè ha tasso di risposta maggiore (1.33 > 1.00). va quindi in esecuzione al tempo 3, con la stessa che termina dopo 3 unità di tempo, ossia al tempo 6, con tempo di turnaround pari a 4 unità di tempo (1 di attesa + 3 di servizio). Nel frattempo, al tempo 4, si è aggiunto alla coda anche il processo , che viene schedulato poichè ha tasso di risposta maggiore di (2.00 > 1.60). va quindi in esecuzione al tempo 6, con la stessa che termina dopo 2 unità di tempo, ossia al tempo 8, con tempo di turnaround pari a 4 unità di tempo (2 di attesa + 2 di servizio). Nel frattempo, al tempo 8, si è aggiunto alla coda anche il processo , ma al termine di viene schedulato poiché ha tasso di risposta maggiore rispetto a (2.00 > 1.00). va quindi in esecuzione al tempo 8, con la stessa che termina dopo 5 unità di tempo, ossia al tempo 13, con tempo di turnaround pari a 10 unità di tempo (5 di attesa + 5 di servizio). Al termine di nella coda rimane solo il processo che quindi viene schedulato. va quindi in esecuzione al tempo 13, con la stessa che termina dopo 3 unità di tempo, ossia al tempo 16, con tempo di turnaround pari a 8 unità di tempo (5 di attesa + 3 di servizio).

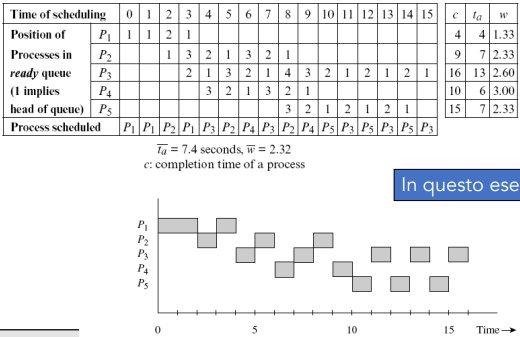
## Scheduling con prelazione

Nello scheduling con prelazione, i processi possono essere prelazionati, in favore di altri processi, prima che possano completare le proprie operazioni. I processi prelazionati vengono ricondotti all’interno della lista di processi ready, in modo da poter continuare la loro esecuzione quando verranno nuovamente schedulati. Una richiesta può perciò essere schedulata diverse volte prima di essere completata; questa caratteristica genera un maggior overhead rispetto allo scheduling senza prelazione. Questo tipo di scheduling viene utilizzano nei SO multi-programmati e time-sharing. Esempi di politiche di scheduling con prelazione sono lo **scheduling Round-Robin con Time-Slice** (**RR**), lo **scheduling Least Completed Next** (**LCN**) e lo **scheduling Shortest Time to Go** (**STG**). Andremo Immagine che contiene testo, orologio

Descrizione generata automaticamenteora a vedere come funzionano queste politiche di scheduling, a partire dai processi

### Scheduling Round-Robin con Time-Slice (RR)

L’idea di questa politica è quella di **assegnare ad ogni processo un intervallo di tempo**, detto **quanto** (**time slice**), durante il quale il processo può utilizzare la CPU. Alla fine di ogni quanto di tempo, il kernel genera un **timer** **interrupt**, in modo da sospendere il processo. Se alla fine del quanto il processo non ha terminato le sue operazioni, allora viene prelazionato ed inserito di nuovo nella coda, con conseguente riassegnazione della CPU ad un altro processo. Se invece il processo si blocca o termina le sue operazioni prima dello scadere del quanto, allora viene effettuata la riassegnazione della CPU ad un altro processo. Questo algoritmo **privilegia i processi CPU bound** perchè utilizzano l’intero quanto assegnato. Il round robin è facile da implementare in quanto lo scheduler deve semplicemente mantenere una coda di processi in stato ready e selezionare il primo processo in coda. Dopodichè, quando il processo scade, questo viene prelazionato e messo in fondo alla coda. Bisogna fare però attenzione alla **scelta della durata del quanto**, poichè un intervallo di tempo **troppo breve provocherebbe** **troppi context switch**, peggiorando l’efficienza della CPU, ma scegliere un **quanto troppo grande** potrebbe provocare **tempi di risposta lunghi** per richieste brevi. Nell’esempio che vedremo ora, il quanto scelto è = 1.



Al tempo 0 e al tempo 1, nella coda dei processi è presente solo che quindi viene mandato in esecuzione ( = 2/3 di servizio). Al tempo 2 arriva il processo , quindi la coda contiene, nell’ordine, , di conseguenza viene eseguito ( = 1/3 di servizio). Al tempo 3 arriva il processo , quindi la coda contiene, nell’ordine, , di conseguenza viene eseguito , che termina le sue operazioni ed esce dalla coda ( = 3/3 di servizio), con tempo di turnaround pari a 4 (1 di attesa + 3 di servizio). Al tempo 4 arriva il processo , quindi la coda contiene, nell’ordine, , di conseguenza viene eseguito ( = 1/5 di servizio). Al tempo 5 la coda contiene, nell’ordine, , di conseguenza viene eseguito ( = 2/3 di servizio). Al tempo 6 la coda contiene, nell’ordine, , di conseguenza viene eseguito ( = 1/2 di servizio). Al tempo 7 la coda contiene, nell’ordine, , di conseguenza viene eseguito ( = 2/5 di servizio). Al tempo 8 arriva il processo , quindi la coda contiene, nell’ordine, , di conseguenza viene eseguito , che termina le sue operazioni ed esce dalla coda ( = 3/3 di servizio), con tempo di turnaround pari a 7 (4 di attesa + 3 di servizio). Al tempo 9 la coda contiene, nell’ordine, , di conseguenza viene eseguito , che termina le sue operazioni ed esce dalla coda ( = 2/2 di servizio), con tempo di turnaround pari a 6 (4 di attesa + 2 di servizio). Al tempo 10 la coda contiene, nell’ordine, , di conseguenza viene eseguito ( = 1/3 di servizio). Al tempo 11 la coda contiene, nell’ordine, , di conseguenza viene eseguito ( = 3/5 di servizio). Al tempo 12 la coda contiene, nell’ordine, , di conseguenza viene eseguito ( = 2/3 di servizio). Al tempo 13 la coda contiene, nell’ordine, , di conseguenza viene eseguito ( = 4/5 di servizio). Al tempo 14 la coda contiene, nell’ordine, , di conseguenza viene eseguito , che termina le sue operazioni ed esce dalla coda ( = 3/3 di servizio), con tempo di turnaround pari a 7 (4 di attesa + 3 di servizio). Al tempo 15 la coda contiene solo , di conseguenza viene eseguito, terminando le sue operazioni ed uscendo dalla coda ( = 5/5 di servizio), con tempo di turnaround pari a 13 (8 di attesa + 5 di servizio).

### Scheduling Least Completed Next (LCN)

In questo tipo di scheduling viene selezionato il **processo che ha utilizzato il minor tempo di CPU**. In caso di uguaglianza, lo scheduler seleziona il **processo che è in attesa da più tempo**. Tutti i processi progrediscono circa di pari passo in termini di tempo di CPU utilizzato; questa politica perciò garantisce che **processi brev**i (I/O bound) **finiscano prima** di processi lunghi (CPU bound). Questo tipo di scheduling inoltre presenta problematiche sia di **starvation** **dei processi lunghi** che di **trascuratezza dei processi esistenti in favore di quelli nuovi**.

Immagine che contiene grafico

Descrizione generata automaticamente

Al tempo 0 e al tempo 1 viene eseguito poichè è l’unico processo a trovarsi in memoria ( = 2/3 di servizio). Al tempo 2 arriva il processo , che viene eseguito immediatamente poichè è quello con tempo di CPU utilizzato minore ( = 1/3 di servizio). Al tempo 3 arriva il processo , che viene eseguito immediatamente poichè è quello con tempo di CPU utilizzato minore ( = 1/5 di servizio). Al tempo 4 arriva il processo , che viene eseguito immediatamente poichè è quello con tempo di CPU utilizzato minore ( = 1/2 di servizio). Al tempo 5 i processi , e hanno lo stesso tempo di CPU utilizzato, di conseguenza viene selezionato , poichè è quello che attende da più tempo ( = 2/3 di servizio). Al tempo 6 i processi e hanno lo stesso tempo di CPU utilizzato, di conseguenza viene selezionato , poichè è quello che attende da più tempo ( = 2/5 di servizio). Al tempo 7 viene eseguito il processo , poichè è quello con tempo di CPU utilizzato minore ( = 2/2 di servizio); inoltre termina la sua esecuzione ed esce, con tempo di turnaround pari a 4 unità di tempo (2 di attesa + 2 di servizio). Al tempo 8 arriva il processo , che viene eseguito immediatamente poichè è quello con tempo di CPU utilizzato minore ( = 1/3 di servizio). Anche al tempo 9 viene eseguito , poichè è quello con tempo di CPU utilizzato minore ( = 2/3 di servizio). Al tempo 10 i processi , e hanno lo stesso tempo di CPU utilizzato, di conseguenza viene selezionato , poichè è quello che attende da più tempo ( = 3/3 di servizio); inoltre termina la sua esecuzione ed esce, con tempo di turnaround pari a 11 unità di tempo (8 di attesa + 3 di servizio). Al tempo 11 i processi , e hanno lo stesso tempo di CPU utilizzato, di conseguenza viene selezionato , poichè è quello che attende da più tempo ( = 3/3 di servizio); inoltre termina la sua esecuzione ed esce, con tempo di turnaround pari a 10 unità di tempo (7 di attesa + 3 di servizio). Al tempo 12 i processi e hanno lo stesso tempo di CPU utilizzato, di conseguenza viene selezionato , poichè è quello che attende da più tempo ( = 3/5 di servizio). Al tempo 13 viene eseguito il processo , poichè è quello con tempo di CPU utilizzato minore ( = 3/3 di servizio); inoltre termina la sua esecuzione ed esce, con tempo di turnaround pari a 6 unità di tempo (3 di attesa + 3 di servizio). Al tempo 14 e al tempo 15, nella coda, è presente solo , che quindi completa le sue operazioni ( = 5/5 di servizio) con tempo di turnaround pari a 13 unità di tempo (8 di attesa + 5 di servizio).

### Scheduling Shortest Time to Go (STG)

Immagine che contiene grafico

Descrizione generata automaticamenteQuesta politica di scheduling seleziona il **processo che ha il minimo requisito di tempo di CPU restante**, di conseguenza **privilegia i processi più brevi**, e fornisce un **buon throughput**. Vengono favoriti, inoltre, i **processi in prossimità di completamento**, **rispetto a nuovi processi brevi** che arrivano nel sistema. Questa caratteristica aiuta a **migliorare i tempi di turnaround e turnaround pesato**. Lo **svantaggio** di questa politica, come per la politica SJF (l’intero **algoritmo** è molto **simile a quello della politica SJF**), sta nel fatto che i processi lunghi possono andare incontro a **starvation**.

Come abbiamo detto in precedenza, vengono eseguiti i processi in ordine di minor requisito di tempo di CPU restante. Inoltre in questo sistema si assume che un processo ammesso nello stesso istante in cui ne viene sospeso un altro sia inserito nella coda prima del processo sospeso. Al tempo 0,1 e 2 viene eseguito il processo , al tempo 0 e al tempo 1 poichè era l’unico processo nella coda, mentre al tempo 2 perchè ha minor requisito di tempo di CPU restante rispetto a (1 < 3), che è arrivato nella coda al tempo 2. termina quindi la sua esecuzione ed esce dalla coda, con tempo di turnaround pari a 3 unità di tempo (0 di attesa + 3 di servizio). Al tempo 3 arriva nella coda dei processi , ma viene eseguito ( = 2 di servizio rimanente) poichè ha minor requisito di tempo di CPU restante rispetto a (3 < 5). Al tempo 4 arriva nella coda dei processi anche , che ha stesso requisito di tempo di CPU restante, ma essendo appena arrivato viene favorito rispetto a e quindi selezionato ( = 1 di servizio rimanente). Al tempo 5 viene nuovamente scelto ( = 0 di servizio rimanente), poiché ha il minimo requisito di tempo di CPU restante (1 < 2 e 1 < 5). termina quindi la sua esecuzione ed esce dalla coda, con tempo di turnaround pari a 2 (0 di attesa + 2 di servizio). Al tempo 6 e al tempo 7 viene selezionato ( = 0 di servizio rimanente), poichè ha minor requisito di tempo di CPU restante rispetto a (2 < 5 e 1 < 5). Al tempo 8 arriva nella coda dei processi , che viene eseguito immediatamente e anche ai successivi tempi 9 e 10, poichè ha minor requisito di tempo di CPU restante rispetto a (3 < 5, 2 < 5 e 1 < 5). termina quindi la sua esecuzione ed esce dalla coda, con tempo di turnaround pari a 3 (0 di attesa + 3 di servizio). A questo punto nella coda rimane solo il processo , che viene eseguito fino al suo completamento. Al tempo 16, quindi, viene completato ed esce dalla coda, con un tempo di turnaround pari a 13 unità di tempo (8 di attesa + 5 di servizio).

## Scheduling in pratica

Per fornire una combinazione appropriata di prestazioni di sistema e servizio utente, il SO deve adattare il suo funzionamento alla natura e al numero delle richieste utente, e alla disponibilità delle risorse. Un singolo scheduler che utilizza una strategia di scheduling classica non può affrontare in modo efficace tutti questi problemi, perciò i **SO moderni impiegano più scheduler** (fino a 3). Alcuni scheduler, inoltre, possono utilizzare una **combinazione di diverse strategie di scheduling**.

## Scheduler a lungo, medio e breve termine

Esistono **vari tipi di scheduling**:

* **lungo termine**, dove lo scheduler decide quale processo entra nella coda dei processi ready tra quelli che la richiedono. Uno scheduler a lungo termine può ritardare l’ammissione di una richiesta nel caso in cui non riesca ad allocare risorse sufficienti o nel caso in cui l’ammissione portasse ad un calo delle prestazioni del sistema;
* **medio termine**, nel quale lo scheduler gestisce la permanenza in memoria dei processi non in esecuzione. In pratica decide quali processi possono risiedere in memoria e quali sul disco, ma le operazioni (**swap in** e **swap out**) vengono comunque effettuate dal gestore della memoria;
* **breve termine**, dove lo scheduler seleziona un processo da una lista di processi **ready** e lo manda al meccanismo di **dispatching**. Può inoltre decidere quanto tempo un processo può utilizzare la CPU e istruisce il kernel affinchè produca un **timer** **interrupt** di conseguenza;
* **I/O**, nel quale lo scheduler gestisce le richieste di I/O pendenti dei processi. In particolare ogni richiesta di I/O pendente deve essere gestita da un dispositivo di I/O disponibile.

## Scheduling con priorità

Una soluzione efficiente per quanto riguarda i dati di scheduling relativi allo scheduling basato su priorità è quella di **mantenere una lista separata di processi ready per ogni valore di priorità**. Questa lista è organizzata come una **coda di PCB**, in cui un PCB punta al PCB del successivo processo in coda. La testa della coda contiene **due puntatori**: uno punta al **PCB del primo processo** nella coda, e l’altro punta alla **testa della coda per la priorità inferiore successiva**. Lo scheduler scandisce le teste in ordine di priorità decrescente e seleziona il primo processo nella prima coda non vuota che trova. In questo modo l’**overhead** dello scheduling dipende dal numero di priorità distinte, piuttosto che dal numero di processi ready.

Immagine che contiene schermata, testo, diagramma, linea

Descrizione generata automaticamente

Lo scheduling basato su priorità può condurre alla **starvation dei processi a bassa priorità**, che può essere **risolto tramite** la tecnica dell’**aging dei processi**, la quale incrementa la priorità di un processo ready se esso non viene selezionato per un certo periodo di tempo. In questo caso, ovviamente, le **priorità** dei processi sarebbero **dinamiche** e il PCB di un processo dovrebbe essere spostato tra diverse code di processi ready. La starvation nello scheduling basato su priorità può causare anche una situazione chiamata **inversione di priorità**. Infatti se un processo ad altà priorità necessita di una risorsa attualmente assegnata ad un processo a bassa priorità, quest’ultimo andrebbe incontro a starvation, e quindi il processo ad alta priorità rimarrebbe blocked indefinitamente. Questa situazione si risolve utilizzando il **protocollo di ereditarietà della priorità** (**priority inheritance protocol**), il quale **aumenta temporaneamente la priorità** del processo a bassa priorità a quella del processo ad alta priorità, in modo che il primo possa utilizzare la risorsa, per poi rilasciarla. Quando il processo rilascia la risorsa, il kernel riporta la sua priorità al precedente valore.

## Message Passing (Scambio di Messaggi)

Lo scambio di messaggi si adatta a diverse situazioni in cui lo scambio di informazioni tra processi gioca un ruolo fondamentale. Uno dei suoi usi più importanti è nel modello **client-server**, in cui un processo **server** fornisce un servizio, e altri processi, chiamati **client**, gli inviano messaggi per utilizzare tale servizio. Un altro importante utilizzo dello scambio di messaggi è nei protocolli di alto livello per lo scambio di email. Quando i processi interagiscono tra loro devono soddisfare **due requisiti fondamentali**: devono essere **sincronizzati** per garantire la mutua esclusione ed hanno bisogno di **scambiarsi** **informazioni** per cooperare tra loro. Lo scambio di messaggi è appunto un approccio per soddisfare entrambe le necessità.

### Send e receive

Il message passing è solitamente implementato mediante le primitive **send** (destination, message) e **receive** (source, message). Un processo invia le informazioni sotto forma di messaggio verso un altro processo designato come destinazione (send), mentre un processo riceve informazioni indicando la sorgente e il messaggio (receive). Per scambiarsi messaggi due processi devono essere sincronizzati, infatti non è possibile che un processo riceva un messaggio prima che il mittente lo abbia inviato. Quando viene eseguita la primitiva send ci sono **due possibilità**:

* il processo mittente si **blocca** finchè il ricevente non riceve il messaggio;
* il processo mittente **continua** la sua esecuzione subito dopo aver inviato il messaggio

Analogamente, quando è eseguita la primitiva receive da un processo, **può accadere che:**

* **se** **non c’è messaggio** in attesa il processo si blocca in attesa dello stesso, o continua l’esecuzione abbandonando il tentativo di ricezione;
* **se un messaggio** è stato precedentemente inviato, lo stesso è ricevuto e l’esecuzione continua.

Una **send bloccante** blocca il processo mittente finchè il messaggio da inviare non viene consegnato al processo destinatario (**message passing sincrono**), mentre una **send non bloccante** consente ad un mittente di proseguire la propria esecuzione, dopo aver effettuato una chiamata send, senza preoccuparsi dell’immediata consegna del messaggio (**message passing asincrono**). Di conseguenza, se abbiamo send e receive **entrambe bloccanti**, il mittente ed il destinatario sono bloccati fino a quando il messaggio non viene consegnato, permettendo una sincronizzazione stretta tra i processi. Quando la **send è non bloccante**, invece, se anche la **receive è non bloccante**, allora a nessuna delle parti è richiesto di attendere, mentre se la **receive è bloccante**, il mittente continua, ma il ricevitore è bloccato fino a quando non riceve il messaggio. Questa è la combinazione più utile, in quanto un processo invia uno o più messaggi a più destinazioni, il più velocemente possibile. Le relazioni tra mittenti e destinatari possono essere:

* **uno a uno**: esiste un ricevitore a cui è associato un mittente (**comunicazione** **privata** tra due processi);
* **uno a molti**: un mittente a più destinatari (**messaggio broadcast**);
* **molti a uno**: utile per applicazioni di tipo client-server dove si ha un ricevitore e un certo numero di client (la mailbox viene chiamata porta);
* **molti a molti**: trattasi di più processi server che devono fornire più processi client.

## Deadlock

Si ha una situazione di deadlock quando, in un insieme di processi , ogni processo in è bloccato da qualche evento , che può essere causato solo da azioni prodotte da un altro, o più, processi in . Ogni processo dunque è in **attesa di un evento** che potrebbe non verificarsi mai.

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

In questo esempio vediamo che i processi e andranno in deadlock nelle loro rispettive seconde richieste, in quando richiede la stampante, la quale è allocata al processo fin quando quest’ultimo non la rilascia, ma allo stesso tempo richiede l’unità nastro, la quale è allocata al processo fin quando questo non la rilascia. Questo esempio riguarda il **deadlock generato da risorse** (**resource** **deadlock**), ma in un SO si possono verificare anche **altri tipi di deadlock**, come ad esempio quello **di sincronizzazione**, che si verifica quando gli eventi attesi sono sotto forma di segnali tra processi, o anche quello **di comunicazione**, che avviene se ogni processo invia un messaggio solo dopo aver ricevuto un messaggio da qualche altro processo dell’insieme.

### Deadlock nell’allocazione di risorse

Un SO gestisce numerose risorse di diverso tipo, sia hardware che software. Viene utilizzato il termine **unità di risorsa** per identificare una risorsa di un certo tipo e il termine **classe di risorsa** per indicare l’insieme di tutte le unità di risorsa di un certo tipo. Per la gestione delle risorse, il kernel gestisce **una tabella delle risorse** per tener traccia dello stato di allocazione di una risorsa. Questa tabella contiene varie informazioni, come il **nome** della risorsa, la **tipologia**, l’**indirizzo** e lo **stato**. L’allocazione delle risorse in un sistema implica tre tipi di eventi: r**ichiesta di una risorsa,** **effettiva allocazione** della stessa e **rilascio** al termine dell’utilizzo. Al rilascio della risorsa, questa può essere allocata ad un altro processo.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

### Condizioni per un deadlock di risorsa

Un deadlock di risorsa di verifica quando si verificano **quattro condizioni contemporaneamente**:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente.

Oltre a queste condizioni è anche essenziale che **non vi sia alcun ritiro di richieste di risorse**, ossia un processo bloccato su una richiesta non può ritirare la stessa.

### Modelli dello stato di allocazione delle risorse

Per stabilire se un insieme di processi è in deadlock è necessario analizzare sia le **informazioni relative alle risorse allocate** che quelle relative **alle richieste di risorse in attesa**. Queste informazioni costituiscono lo stato di allocazione delle risorse di un sistema, e per rappresentarlo si utilizzano **due tipi di modelli**: **modelli basati su grafo**, dove un processo può richiedere e utilizzare esattamente un’unità di risorsa di ogni classe di risorse, e **modelli basati su matrice**, nei quali un processo può richiedere un qualsiasi numero di unità di una classe di risorse. Nei **modelli basati su grafo** abbiamo:

* **grafo di richiesta e di allocazione di risorsa** (**RRAG**), nel quale un processo può richiedere e utilizzare esattamente **un’unità di risorsa di ogni classe di risorsa** (che però **può avere anche più unità di risorse al suo interno**). In questo grafo esistono due tipi di nodi, i **nodi processi**, rappresentati da **cerchi**, e i **nodi risorsa**, rappresentati da **rettangoli** e che rappresentano classi di risorse. Il **numero di pallini** nei nodi risorsa indica quante unità di quella classe esistono nel sistema. Per collegare i nodi si utilizzano gli **archi**, ovviamente, e anche in questo caso ne abbiamo **due tipi**. Un **arco di allocazione** è diretto **da un nodo risorsa ad un nodo processo**, e indica che la risorsa è assegnata al processo, mentre un **arco di richiesta** è diretto **da un nodo processo ad un nodo risorsa**, e indica che il processo è attualmente bloccato in attesa della risorsa. Un arco di allocazione viene cancellato quando il processo rilascia un’unità di risorsa della classe , mentre un arco di richiesta viene cancellato e viene aggiunto un arco di allocazione quando è concessa una richiesta in attesa dal processo per un’unità di una classe

Immagine che contiene diagramma, schematico

Descrizione generata automaticamente

;

* **grafo di attesa** (**WFG**), che può essere utilizzato per descrivere lo stato delle risorse di un sistema in cui **ogni classe di risorse contiene solo un’unità di risorsa**. Nel WFG esiste **un solo tipo di nodi**, detti **nodi processo**, e gli **archi** rappresentano la relazione **wait-for** tra processi, che indica che un processo è bloccato su una richiesta per una risorsa attualmente allocata al processo , con il primo che in pratica attende che il secondo la rilasci.

Quando ogni classe di risorse contiene una sola unità di risorsa, il RRAG e il WFG risultano equivalenti.

Immagine che contiene diagramma, linea, Carattere, cerchio

Descrizione generata automaticamenteUn **cammino in un grafo** è una sequenza di archi tali che il nodo destinazione di un arco è il nodo sorgente dell’arco seguente. Consideriamo un cammino in un RRAG : questo indica che al processo è stata allocata un’unità di , al processo è stata allocata un’unità di ed aspetta un unità di ecc. Nel WFG, lo stesso cammino sarebbe .

Se supponiamo che ogni classe di risorsa contenga un’unica unità di risorsa, i due cammini appena visti sono privi di deadlock. **Se un grafo non contiene cicli, allora non può esistere un deadlock**.

Immagine che contiene cavo, cerchio, linea

Descrizione generata automaticamente

Un **ciclo in un RRAG**, inoltre, **non implica necessariamente un deadlock** se una classe di risorse ha più di un’unità di risorsa.

Immagine che contiene cavo, diagramma, linea

Descrizione generata automaticamente

Nel **modello a matrice**, invece, **un processo può richiedere un qualsiasi numero di unità di una classe di risorsa**. Lo stato di allocazione di un sistema viene rappresentato da **due matrici**:

* la **matrice delle risorse allocate**, che indica quante unità di risorsa di ogni classe di risorse sono allocate ad ogni processo del sistema;
* la **matrice delle risorse richieste**, che invece tiene conto delle richieste in attesa, e indica quante unità di risorsa di ogni classe di risorse sono state richieste da ogni processo del sistema.

Se consideriamo un sistema con processi e classi di risorse, le due matrici appena descritte avranno dimensione . In alcuni casi vengono utilizzate anche **tabelle ausiliarie** per rappresentare informazioni aggiuntive, come ad esempio la **matrice delle risorse totali** e la **matrice delle risorse disponibili**, entrambe matrici colonna con elementi. La prima indica il numero totale di unità di risorse in ogni classe di risorse, mentre la seconda indica il numero di unità di risorse di ogni classe di risorse attualmente disponibili.

## Gestione dei deadlock

Per gestire i deadlock esistono **tre approcci fondamentali**, ognuno con diversi comportamenti in termini di possibili ritardi nell’allocazione delle risorse, di tipo di richieste di risorse che i processi utente possono effettuare e di overhead di SO.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

### Individuazione (rilevamento) e risoluzione dei deadlock

Questo approccio non tenta di impedire il verificarsi di un deadlock, ma al contrario controlla direttamente se si sono verificati, per poi trovare un modo per risolvere il problema. Un processo bloccato non è coinvolto in un deadlock se la richiesta su cui è bloccato può essere soddisfatta con la sequenza di eventi **completamento processo – rilascio risorsa – allocazione risorsa**. Se ogni classe di risorse nel sistema contiene una sola unità di risorsa, allora il controllo su possibili deadlock avviene **verificando la presenza di cicli** nel RRAG o nel WFG. Questa soluzione non è ammissibile nel caso di classi di risorse contenenti più unità di risorse, in quanto il WFG non potrebbe essere utilizzato, mentre il **RRAG** avrebbe bisogno di **algoritmi piuttosto complessi** per essere utilizzato. Proprio per questo viene utilizzato il modello a matrice, in quanto applicabile in tutte le situazioni. Si cerca di costruire una **sequenza ammissibile di eventi**, dove tutti i processi bloccati possono ottenere le risorse che hanno richiesto. Se il tentativo va a buon fine allora non c’è deadlock, **nel caso in cui fallisca** ovviamente **significa che c’è deadlock**. Per spiegare meglio come funziona l’individuazione dei deadlock, consideriamo un **esempio** in cui abbiamo il seguente stato di allocazione di un sistema con 10 unità di una classe di risorse e tre processi :

Immagine che contiene testo, Carattere, diagramma, bianco

Descrizione generata automaticamente.

Il processo è in stato running perchè non è bloccato da una richiesta di risorse. Tutti i processi nel sistema possono terminare se: termina e rilascia le 2 unità della risorsa ad esso allocato, con conseguente allocazione delle stesse a . Quando anche questo processo termina, 6 unità della risorsa possono essere allocate a . In questo modo non c’è alcun processo bloccato quanto termina la simulazione e di conseguenza non c’è deadlock nel sistema.

Andiamo ora a vedere un **esempio di algoritmo di individuazione deadlock**:

Immagine che contiene testo, Carattere, lettera, documento

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, bianco, ricevuta

Descrizione generata automaticamenteL’algoritmo prende in input due insiemi di processi, **Blocked** e **Running**, e un modello basato su matrice dello stato di allocazione, comprendente le matrici **Risorse\_allocate**, **Risorse\_richieste** e **Risorse\_libere**. L’algoritmo simula il completamento di un processo running , trasferendolo dall’insieme Running all’insieme **Exit**. Le risorse allocate per vengono quindi aggiunte alle risorse libere e l’algoritmo procede a selezionare un processo dall’insieme Blocked la cui richiesta di risorse può essere soddisfatta dalle risorse attualmente libere, trasferendo questo processo all’insieme Running. Successivamente l’algoritmo simula il suo completamento e lo trasferice da Running a Exit. L’algoritmo itera queste operazioni fino a quando non è rimasto alcun processo nell’insieme Running. Se rimangono processi nell’insieme Blocked, allora significa che essi sono in deadlock. La complessità dell’algoritmo si ricava dalla dimensione massima degli insieme Running e Blocked, che possono contenere fino a processi, con il numero totale dei processi del sistema. Il loop del passo 1 itera al più n volte e il passo 1(d) esegue un numero di operazioni dell’ordine ad ogni iterazione. Di conseguenza l’algoritmo richiede una **complessità dell’ordine** . Consideriamo ora un sistema con quattro processi ,,, e 5, 7, 5 unità nelle classi di risorsa rispettivamente. Lo stato, prima che effettui una richiesta di un’unità della classe di risorsa , è il seguente:

Il primo passo da effettuare è quello di assegnare una unità di risorsa della classe al processo , come precedentemente detto, e si utilizza l’algoritmo appena visto per l’individuazione dei deadlock. Gli insiemi Blocked e Running, che fungono da input dell’algoritmo, sono inizializzati rispettivamente a e , mentre le matrici Risorse\_allocate, Risorse\_richieste e Risorse\_libere sono mostrate nella figura precedente. L’algoritmo trasferisce quindi il processo nell’insieme Exit e libera le risorse ad esso allocate. Il numero delle unità libere delle classi di risorsa è ora 1, 1 e 2, rispettivamente. L’algoritmo verifica che ora può essere soddisfatta la richiesta in attesa del processo , quindi gli alloca le risorse richieste e trasferisce nell’insieme Running. Dopo aver liberato le risorse di , l’algoritmo verifica che ora può essere soddisfatta la richiesta di risorsa di e, dopo che termina, può essere soddisfatta la richiesta di risorsa di . Ora l’insieme Running è vuoto, l’algoritmo termina e nel sistema non esistono deadlock in quanto l’insieme Blocked risulta vuoto.

Immagine che contiene testo, ricevuta, modello

Descrizione generata automaticamenteNel caso venga individuato un insieme di processi in stato di deadlock, bisogna passare alla fase di **risoluzione deadlock**, che comporta l’interruzione del deadlock per assicurare il progresso di alcuni processi di . Ciò si ottiene forzando la terminazione di uno o più processi di e allocando le loro risorse a qualche altro processo. Ogni processo terminato è detto **vittima** della risoluzione del deadlock. La scelta della vittima viene fatta in base a criteri quali priorità di un processo, risorse consumate, ecc. Andiamo ora a vedere un **esempio di risoluzione deadlock**:

Immagine che contiene diagramma, linea, schizzo

Descrizione generata automaticamente

Il RRAG in figura (a) mostra un deadlock che coinvolge i processi ,,,. Questo deadlock è risolto scegliendo il processo come vittima. La parte (b) della figura mostra il RRAG dopo la terminazione del processo e l’allocazione al processo della risorsa , precedentemente allocata a . Il processo , che era in attesa della vittima prima della risoluzione del deadlock, ora attende , che è il nuovo possessore della risorsa. Se lo stato di allocazione è rappresentato dal modello basato su matrice, è sufficiente cancellare le righe corrispondenti a in Risorse\_allocate e Risorse\_richieste, modificare le righe del processo e modificare Risorse\_libere di conseguenza.

### Prevenzione dei deadlock

Come abbiamo già visto, per verificarsi un deadlock devono verificarsi contemporaneamente **quattro condizioni**. Per prevenire i deadlock, il kernel deve utilizzare una **politica di allocazione di risorse** che assicuri che una di queste condizioni non possa verificarsi. Queste politiche di allocazione di risorse, che vedremo in seguito, utilizzano differenti **approcci alla prevenzione dei deadlock**:

* **condivisione di risorse**, infatti in un sistema non esisterebbero deadlock se **tutte le risorse** potessero essere rese **condivisibili**. In questo modo in un RRAG ci sarebbero **solo archi di allocazione**, quindi non si avrebbero mai attese circolari. Il problema è che **alcune risorse non sono condivisibili**, come le stampanti, ma si può aggirare tale problema creando dispositivi virtuali, come ad esempio una stampante virtuale da allocare a vari processi. Questo approccio però non può funzionare per risorse software come file condivisi, che dovrebbero essere modificati in mutua esclusione per evitare race condition.

Immagine che contiene testo, schermata, diagramma, bianco

Descrizione generata automaticamente

;

* **prelazione di risorse**, se le **risorse** fossero **prelazionabili**, il kernel potrebbe assicurare che i processi avrebbero tutte le risorse di cui necessitano, il che **evita percorsi circolari nel RRAG**. L’unico problema è che i dispositivi sequenziali di I/O non possono essere prelazionati.

Immagine che contiene Carattere, diagramma, bianco, schizzo

Descrizione generata automaticamente

;

* Immagine che contiene testo, diagramma, schermata, Carattere

  Descrizione generata automaticamente**hold and wait**, per evitare la condizione relativa al possesso e attesa, è necessario che un processo che abbia acquisito risorse **non possa effettuare richieste di risorse**, oppure che un processo bloccato su una richiesta di risorsa non possa impegnare altre risorse. In un RRAG, i percorsi che coinvolgono più di un processo non possono comparire, quindi **non possono esistere percorsi circolari**. Una semplice politica per l’implementazione di questo approccio è consentire ad un processo di effettuare solo una richiesta di risorse durante la propria esecuzione, in cui chiede tutte le risorse di cui necessita.

;

* Immagine che contiene diagramma, Carattere, bianco, linea

  Descrizione generata automaticamente**attesa circolare**, la quale può dipendere dalla condizione hold and wait, che a sua volta è conseguenza delle condizioni di mutua esclusione e assenza di prelazione; quindi **non si verifica se non si verifica nessuna di queste condizioni**. Le attese circolari possono essere evitate separatamente non consentendo ad alcuni processi di attendere determinate risorse.

Per quanto riguarda le politiche di prevenzione dei deadlock abbiamo l’allocazione simultanea e il ranking delle risorse:

* **allocazione simultanea**, che è la più semplice tra le politiche di prevenzione. Un processo deve richiedere **tutte le risorse** di cui ha necessità **con un’unica richiesta**, in modo che il kernel gliele allochi tutte insieme. In questo modo ai processi blocked non vengono assegnate risorse, quindi la condizione hold and wait non è mai soddisfatta e di conseguenza non si possono verificare attese circolari e deadlock. La semplicità di implementazione rende questa politica interessante per **piccoli SO**, ma pregiudica l’efficienza delle risorse in quanto, ad esempio, se una risorsa viene utilizzata verso la fine dell’esecuzione di un processo , dovendola richiedere comunque all’inizio, ritarda gli altri processi che magari arrivano dopo l’inizio di ma utilizzerebbero prima quella risorsa. Questa situazione quindi **riduce il grado di multiprogrammazione**, il quale **pregiudica l’efficienza della CPU**;
* **ranking delle risorse**, che utilizza un numero identificativo, chiamato **rank di risorsa**, associato ad ogni classe di risorsa. Quando un processo effettua una richiesta di una risorsa, il kernel applica un **vincolo di validità** per decidere se la richiesta deve essere considerata. Il vincolo di validità assume il valore vero solo se il rank della risorsa richiesta è maggiore del rank massimo della risorsa attualmente allocata per . In questo caso, la risorsa è allocata a se è disponibile, altrimenti viene bloccato in attesa che la risorsa venga rilasciata. Se il vincolo di validità è falso, invece, la richiesta viene rifiutata e il processo sarà interrotto. L’assenza di attesa circolare in un sistema di questo tipo può essere spiegata nel seguente modo: sia il rank assegnato alla classe di risorsa e sia il processo a cui sono allocate alcune unità della classe di risorsa . può essere bloccato su una richiesta per una unità di una classe di risorsa solo se . Consideriamo ora il processo , a cui sono allocate alcune unità della classe di risorsa poiché . Quindi, se può attendere che rilasci la risorsa, non può attendere . Di conseguenza i due processi non possono trovarsi in una condizione di attesa circolare. La politica del ranking delle risorse funziona al meglio quando tutti i processi richiedono le risorse in ordine crescente di rank. Le difficoltà nascono quando un processo richiede una risorsa con rank minore, in quanto l’unico modo in cui può ottenere questa risorsa è rilasciando prima la risorsa con rank maggiore. Nel **caso** **peggiore**, tra l’altro, la strategia può degenerare nella strategia di allocazione simultanea. Nonostante questi svantaggi, la politica del ranking delle risorse è interessante per la sua semplicità. Il kernel può utilizzarla per le sue richieste di risorse quando ha bisogno di risorse in un ordine prefissato.

### Evitare i deadlock (Algoritmo del banchiere)

La politica di evitare i deadlock ammette una richiesta di risorsa solo se si riesce a stabilire che l’accettazione della richiesta non può condurre ad un deadlock nè immediatamente nè successivamente; in caso contrario è messa in attesa finchè non viene ammessa. Questa soluzione è possibile solo se il sistema è capace di mantenere delle informazioni sulle risorse disponibili al sistema e sulle risorse che ogni processo può potenzialmente richiedere. Per fare ciò, si utilizza un **approccio** **conservativo**, che consiste nel far dichiarare ad ogni processo il massimo numero di unità di risorsa di ogni classe che può richiedere. Il kernel permette che un processo richieda queste unità di risorsa in passi successivi, ossia pochè unità alla volta, in base al massimo numero dichiarato dal processo. Il kernel, inoltre, utilizza una **tecnica di analisi del caso peggiore** per controllare la possibilità di deadlock successivi. Questo approccio è definito conservativo in quanto un processo può terminare il suo funzionamento senza richiedere il massimo numero dichiarato di unità. Uno degli algoritmi che adotta tale approccio è l’algoritmo del banchiere. Tuttavia, per la maggior parte dei sistemi, è impossibile conoscere in anticipo le risorse che richiederà un processo, per cui è spesso impossibile evitare del tutto i deadlock. L’**algoritmo del banchiere** è un algoritmo utilizzato per prevenire i deadlock nell’allocazione delle risorse. In particolare questo algoritmo può indicare se un sistema si venga a trovare in uno stato sicuro o meno nel caso assegnasse una risorsa ad uno dei processi richiedenti. Uno **stato di allocazione sicuro** è uno stato di allocazione in cui è possibile costruire una sequenza di eventi **completamento processo – rilascio risorsa – allocazione risorsa**. Quando un sistema è in questo stato, tutti i processi possono completare le proprie operazioni senza la possibilità di deadlock. Al contrario, se il sistema è in uno **stato non sicuro**, non vi è nessuna garanzia di questo tipo. Nella tabella che segue vengono descritte le **strutture dati** di tipo array **utilizzate nell’algoritmo del banchiere**:

Immagine che contiene testo, ricevuta, Carattere, schermata

Descrizione generata automaticamente.

In questo algoritmo, la nozione di stato di allocazione sicuro fa sì che ogni processo nel sistema possa ottenere risorse per ogni classe di risorsa , in modo da completare le sue operazioni. Le tecniche per evitare i deadlock sono implementate trasferendo il sistema da uno stato di allocazione sicuro ad un altro stato di allocazione sicuro, come descriveremo a seguire:

1. quando un processo effettua una richiesta, si calcola il nuovo stato di allocazione in cui si troverebbe il sistema se la richiesta fosse ammessa, chiamato **stato di allocazione proiettato**;
2. se lo stato di allocazione proiettato è uno stato di allocazione sicuro, si ammette la richiesta aggiornando le matrici Risorse\_allocate e Risorse\_totali, altrimenti si tiene la richiesta in attesa. La sicurezza dello stato di allocazione proiettato viene verificata con una **simulazione**, ossia si cerca di costruire una sequenza completamento processo – rilascio risorsa – allocazione risorsa. E’ assunto che un processo completi le sue operazioni solo se può prendere il massimo richiesto di ciascuna risorsa contemporaneamente, ovvero per tutti i , ossia se

. Se questa condizione viene soddisfatta, allora viene effettivamente simulato il completamento del processo . In seguito viene controllato se qualche altro processo può soddisfare questa condizione e così via;

1. quando un processo rilascia una qualsiasi risorsa o terminano le operazioni, si esaminano le richieste pendenti e si allocano quelle che pongono il sistema in un nuovo stato sicuro.

Esempio di algoritmo del banchiere per una sola classe di risorse:

prendiamo in considerazione un sistema contenente 10 unità di risorse ( - ) di una singola classe , le cui richieste massime () di risorse effettuate da parte di tre processi e , sono rispettivamente 8, 7 e 5 unità di risorsa, mentre l’allocazione corrente () è 3, 1 e 3.

Immagine che contiene testo, Carattere, bianco

Descrizione generata automaticamente

Da questa situazione, il processo effettua una richiesta di un’unità di risorsa, quindi, nello stato di allocazione proiettato, passa da 7 a 8, con due unità che restano libere. La sicurezza dello stato proiettato è garantita dal fatto che soddisfa la condizione

,

in quanto è esattamente sotto di due unità rispetto alle sue richieste massime. Queste due unità possono essere allocate a se successivamente le richiede, in modo da farlo terminare. A quel punto sarebbero disponibili 5 unità di risorsa, quindi le richieste di di 4 unità sarebbero soddisfatte e potrebbe terminare. Alla terminazione di sarebbero disponibili tutte le unità di risorsa, di conseguenza anche potrebbe terminare. Fatta questa analisi, l’algoritmo accetterà la richiesta effettuata da . La nuova allocazione per i processi è 4, 1 e 3 unità di risorsa e = 8. Consideriamo ora le seguenti richieste:

* fa una richiesta per 2 unità di risorsa;
* fa una richiesta per 2 unità di risorsa;
* fa una richiesta per 2 unità di risorsa.

Come abbiamo visto nello stato di allocazione proiettat in precedenza, può essere accettata solo la richiesta di . Passiamo ora al vero e proprio **algoritmo del banchiere per un qualsiasi numero** **di classi di risorse**:

Immagine che contiene testo, lettera, carta, menu

Descrizione generata automaticamente

Quando un processo effettua una Nuova\_richiesta, questa viene inserita nella matrice Risorse\_richieste, che memorizza le richieste in attesa di tutti i processi. Viene quindi invocato l’algoritmo con l’**identificativo del processo richiedente**. Quando un processo rilascia alcune risorse allocate per sè, oppure termina le proprie operazioni, viene invocato l’algoritmo per ogni processo la cui richiesta è in attesa. L’algoritmo può essere descritto come segue: dopo alcune inizializzazioni al **passo 1**, l’algoritmo simula l’accettazione della richiesta al **passo 2**, calcolando lo stato di allocazione proiettato. Il **passo 3** controlla se lo stato di allocazione proiettato è **fattibile**, ossia se esistono sufficienti risorse libere per permettere l’accettazione della richiesta. Per controllare se lo stato di allocazione proiettato è uno stato sicuro, si controlla se il massimo numero di risorse di cui necessita ogni processo attivo, ossia facente parte degli insiemi **Running** o **Blocked**, può essere soddisfatto allocando alcune delle **risorse libere**. In caso affermativo, l’algoritmo simula il suo completamento cancellandolo dall’insieme dei processi **Active** e rilasciando le risorse allocate per esso. Questa azione viene ripetuta finchè non è possibile cancellare più processi dall’insieme Active. Se al termine di questo passo l’insieme Active è vuoto, lo stato di allocazione proiettato è uno stato sicuro, quindi l’algoritmo cancella la richiesta dalla lista delle richieste in attesa e alloca le risorse richieste. Questa azione non sarà eseguita se lo stato proiettato non è nè fattibile nè sicuro, per cui la richiesta rimane in attesa. Il **passo 4** è simile all’algoritmo di **individuazione dei deadlock**, di conseguenza l’algoritmo del banchiere ha una complessità dell’ordine di . Prendiamo ora in esame un **esempio** che mostra il funzionamento dell’algoritmo del banchiere in un sistema con quattro processi e , con quattro classi di risorsa che contengono 6,4,8 e 5 unità di risorsa, di cui 5,3,5 e 4 unità di risorsa attualmente allocate.

Immagine che contiene testo, documento, ricevuta, modello

Descrizione generata automaticamente

Il processo ha effettuato una richiesta (0,1,1,0), che sta per essere elaborata. L’algorito simula l’accettazione di questa richiesta nel passo 2, e controlla se lo stato di allocazione proiettato è sicuro nel passo 4. In questo stato sono disponibili 1,0,2 e 1 unità di risorsa, quindi solo il processo può terminare, poichè ha bisogno di 1,0,1 e 0 unità di risorse. Di conseguenza l’algoritmo simula il suo completamento, le per vengono rilasciate e lo stesso processo viene cancellato dall’insieme Active. Il processo ha bisogno di 0,1,3 e 0 unità di risorsa per soddisfare la sua richiesta massima di risorse necessarie, quindi ora può avere queste risorse allocate, e di conseguenza terminare. I processi rimanenti possono terminare nell’ordine ; di conseguenza la richiesta effettuata dal processo viene accettata. Un'altra tipologia di **algoritmo del banchiere** è quella **con** **allocazioni parametriche**: si supponga infatti che in un sistema multiprocessore a 3 processori siano presenti cinque processi , allocati sul processore 0, , allocato sul processore 1, e , allocati sul processore 2 e si supponga esista un insieme di risorse di quattro tipi diversi A,B,C e D. Ci si trova nella seguente configurazione:

Immagine che contiene testo, Carattere, numero, schermata

Descrizione generata automaticamente.

In questo caso bisogna **determinare gli intervalli dei valori interi di X,Y e** **Z** per i quali il **sistema si trova in uno stato sicuro**, elencando eventuali sequenze sicure, e se la richiesta attuale di (2,0,0,2) può essere soddisfatta. Inizialmente, l’unico processo che soddisfa la condizione

è , ed in particolare

Immagine che contiene testo, Carattere, calligrafia, bianco

Descrizione generata automaticamente.

Dopo che termina l’esecuzione, lo stesso rilascia le sue risorse e le risorse disponibili diventano [6,X+1,13,6]. A questo punto può essere soddisfatta solo la richiesta di , e le risorse disponibili al termine dello stesso processo sono [6,X+1,16,8]. Ora si può soddisfare solo la richiesta di , e le risorse disponibili al suo termine diventano [10,X+1,16,8]. Si può quindi soddisfare la richiesta di se

Immagine che contiene testo, Carattere, calligrafia, bianco

Descrizione generata automaticamente.

Al rilascio delle risorse di , dopo che è terminato, le risorse disponibili diventano [12,X+2,Z+17,12]. Possono quindi essere soddisfatte le richieste di se

.

In questo caso, al valore di Z, abbiamo sostituito il massimo dei valori che Z può ammettere, ossia 5. In definitiva, la sequenza sicura è , con . Inoltre la richiesta di (2,0,0,2) non può essere soddisfatta inizialmente, ma solo dopo che e hanno rilasciato le loro risorse.

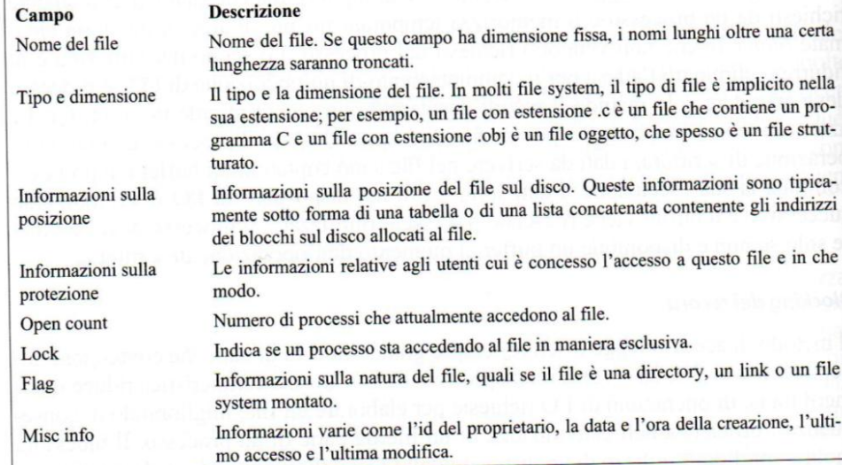
## File System

Tutte le applicazioni hanno bisogno di memorizzare e rintracciare informazioni e dati. Durante l’esecuzione, un processo può memorizzare nella RAM solo una parte di queste informazioni, mentre la restante parte viene memorizzata su dischi e/o supporti esterni. La parte del SO che si occupa dei file prende il nome di file system. Gli obiettivi principali di un file system sono l’**accesso conveniente e veloce ai file,** la **memorizzazione affidabile** **dei file** e la **condivisione dei file** **con altri utenti**. Per raggiungere questi obiettivi in maniera efficace, il file system è strutturato in **due strati**: i **moduli del file system**, che si occupano della condivisione, della protezione e dell’affidabilità dei file, e l’**IOCS** (Input-Output Control System), che si occupa dell’implementazione delle operazioni sui file. Un file system consiste di **due tipi di dati**:

* **dati nel file** (**dati**), ossia quelli contenuti nei file;
* **dati di controllo** (**metadati**), cioè quelli utilizzati per accedere ai file.

## Directory

Un elemento molto importante dei file system sono le directory, che contengono **informazioni relative ad un gruppo di file**. Ogni elemento in una directory contiene gli **attributi di un file**, i cui **campi** **tipici** sono:

.

Una directory contiene i file appartenenti a diversi utenti, quindi deve garantire **due** importanti **prerogative**:

* **libertà nella scelta del nome**, ossia gli utenti devono avere la possibilità di assegnare nomi ai file liberamente;
* **condivisione dei file**, cioè gli utenti devono avere la possibilità di accedere ai file creati da altri utenti e di concedere ad altri utenti il permesso di accedere ai propri file.

Il file system crea tante directory e per organizzarle utilizza una **struttura a due livelli**:

* in cima abbiamo la **directory master**, che contiene informazioni relative alle directory utente di tutti gli utenti del sistema. Ogni elemento di una directory master è una **coppia** formata da un **ID utente** e da un **puntatore ad una directory utente**;
* al secondo livello abbiamo le **directory utente** (**UD**), che contengono elementi utili alla descrizione dei file appartenenti ad un utente.

Immagine che contiene schizzo, diagramma, schermata, cerchio

Descrizione generata automaticamente

Questa soluzione richiede che i nomi siano unici solo all’interno dell’area del singolo utente, infatti l’utilizzo di UD separate permette la **libertà nell’assegnazione di un nome** ad un file. Questo tipo di organizzazione garantisce l’accesso al file corretto anche se nel sistema esistono molti file con lo stesso nome. L’utilizzo delle UD ha però uno **svantaggio**, infatti impedisce agli utenti di condividere i loro file con altri utenti. Per raggiungere questo scopo sono necessarie istruzioni speciali che permettono a degli utenti di accedere ai file di altri utenti, e per fare ciò bisogna verificare i permessi dei file che si trovano nel campo **protection** **info** (**informazioni sulla protezione**).

## Organizzazione dell’I/O (DMA)

Il sottosistema di I/O ha un percorso ai dati in memoria indipendente. La figura mostra come i dispositivi di I/O sono connessi ai controller di dispositivo, che a loro volta sono connessi al **DMA** (**Direct Memory Access**). Ogni **dispositivo** è **identificato** dalla **coppia** (**controller ID**, **device ID**), questo poichè ogni **controller** ha un **ID numerico unico**, ma anche i dispositivi ad esso connesso ne hanno uno (ID numerico unico). Un’**operazione di I/O** coinvolge:

* operazione da eseguire (read, write, ecc.);
* indirizzo del dispositivo di I/O;
* numero di byte di dati da trasferire;
* indirizzi delle aree di memoria e del dispositivo di I/O che sono coinvolti nel trasferimento dei dati.

La CPU avvia l’operazione di I/O mediante l’esecuzione di un’**istruzione di I/O**, che punta ad un insieme di comandi di I/O che specificano le singole operazioni coinvolte nel trasferimento dati, senza coinvolgere direttamente la CPU. L’esecuzione di tali comandi è compito del DMA, del controller del dispositivo e del dispositivo di I/O, mentre appunto la CPU è libera di fare altro mentre l’operazione di I/O è in atto. **Esempio di operazione I/O di lettura**:

si vogliono leggere i dati memorizzati in un **blocco del disco** (**track\_id**, **block\_id**):

**I/O-init** (controller\_id, device\_id), I/O\_command\_addr,

dove **I/O\_command\_addr** è l’indirizzo di partenza dell’area di memoria che contiene **due comandi**:

1. posiziona la testina del disco sulla traccia track\_id;
2. leggi il record record\_id nell’area di memoria con indirizzo di partenza memory\_add.

Quando viene eseguita un’istruzione di I/O entra in gioco la configurazione chiamata **third party DMA**. In pratica all’esecuzione di un’istruzione di I/O, il controller DMA passa i dettagli dei comandi di I/O al controller del dispositivo di I/O, al quale (controller del dispositivo) vengono consegnati anche i dati, dal dispositivo. Il **trasferimento dei dati da controller del dispositivo a memoria** avviene come segue:

1. il controller del dispositivo invia un segnale DMA request quando è pronto al trasferimento;
2. il DMA, ricevuto il segnale, ottiene il controllo del bus, vi pone l’indirizzo di memoria che partecipa al trasferimento ed infine invia un segnale DMA ack (acknowledgment) al controller del dispositivo;
3. il controller del dispositivo trasferisce i dati verso o dalla memoria;
4. alla fine del trasferimento, il controller DMA genera un interrupt di completamento I/O con codice uguale all’indirizzo del dispositivo.

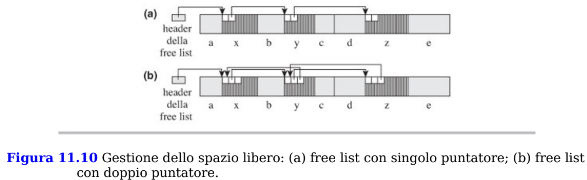
L’**ISR** (**interrupt service routine**) analizza il codice per trovare quale dispositivo ha completato la sua operazione di I/O e intraprende le operazioni appropriate.

## Free list

Una free list è una struttura dati utilizzata per tenere traccia dei blocchi di memoria disponibili. Quando viene allocata memoria per un oggetto o una struttura dati, viene prelevato un o più blocchi dalla free list, mentre quando vengono deallocati, i blocchi di memoria saranno restituiti alla free list. L’utilizzo di una free list consente di evitare sprechi di memoria e di ottimizzare l’allocazione di memoria dinamica, consentendo una gestione efficiente delle risorse di memoria.

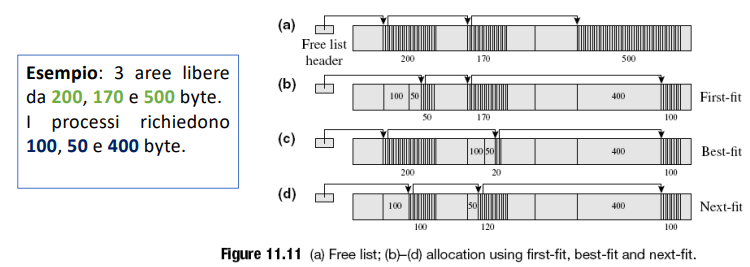
### Gestione delle free list

Il kernel ha bisogno di mantenere **due elementi di informazione di controllo** **per ogni** **area di memoria** nella free list: la **dimensione** dell’area di memoria ed i **puntatori** utilizzati per creare la lista. Per evitare di incorrere in overhead di memoria causato da queste informazioni di controllo, il kernel memorizza queste informazioni in pochi byte **all’inizio della stessa area di memoria libera**.



Nel caso di utilizzo di **un** **solo** **puntatore**, quando si vogliono effettuare operazioni di inserimento o cancellazione, queste vengono eseguite scorrendo la lista a partire dalla testa. Può essere quindi utilizzata una free list con **doppio** **puntatore**, uno che punta all’**area di memoria successiva** e uno che punta all’**area di memoria precedente**, in modo da rendere più efficienti le operazioni di inserimento e cancellazione di aree di memoria. Possono essere utilizzate **tre tecniche** **per** eseguire l’**allocazione della memoria** **utilizzando una free list**:

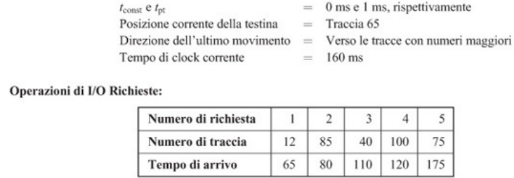
* **first-fit**, nella quale si utilizza la **prima area sufficientemente grande**;
* **best-fit**, dove si usa l’**area sufficientemente grande più piccola**;
* **next-fit**, nella quale viene utilizzata la **successiva area sufficientemente grande**.

****

## Scheduling del disco

Una delle funzioni dell’IOCS e dei driver dei dispositivi è quella di utilizzare lo scheduling del disco per eseguire le operazioni di I/O in un ordine tale da ridurre il movimento delle testine e il tempo medio di attesa delle operazioni di I/O, in modo da ottimizzare il throughput del disco. Al variare del tipo di accesso, chiaramente, saranno diverse le prestazioni del sistema, fermo restando che l’obiettivo comune di tutte le politiche di scheduling è quello di servire il massimo numero di richieste nell’unità di tempo. Esistono **diverse politiche di scheduling del disco**:

* **First-Come First-Served** (**FCFS**), che seleziona l’operazione di I/O con **tempo di richiesta inferiore**;
* **Shortest Seek Time First** (**SSTF**), che seleziona l’operazione di I/O con il **più breve tempo di ricerca** rispetto alla posizione corrente delle testine del disco;
* **SCAN/Look**, in questa politica le testine del disco si muovono **da un’estremità all’altra del piatto**, servendo le operazioni di I/O per i blocchi su ogni traccia/cilindro, prima di spostarsi sulla successiva traccia/cilindro. **Quando** le testine del disco **raggiungono l’altra estremità del piatto**, **la direzione di movimento viene invertita** e le nuove richieste vengono servite nella scansione inversa. Una **variante** della SCAN, chiamata **Look**, **inverte la direzione del disco quando non ci sono più richieste** di I/O **nella** **direzione corrente**;
* **CSCAN/C-Look** (C sta per circolare), in questa politica la scansione è eseguita come nello scheduling SCAN, tuttavia **non viene mai eseguita la scansione inversa**, ma al suo posto vengono spostate **le testine nella posizione di partenza sul piatto** e inizia un’**altra scansione**. La **variante C-Look** **muove** invece **le testine** solo **finchè ci sono richieste da eseguire**, **prima** **di** iniziare una **nuova scansione**.

Immagine che contiene testo, numero, schermata, Carattere

Descrizione generata automaticamente

## Frammentazione della memoria

La frammentazione della memoria è la presenza di a**ree di memoria inutilizzabili nella memoria di un computer**. Questo porta ovviamente ad un utilizzo inefficiente della memoria. Esistono **due tipi** di frammentazione: **frammentazione esterna**, dove alcune aree di memoria sono semplicemente troppo piccole per essere allocate, e **frammentazione interna**, nella quale viene allocata **più memoria rispetto a quella richiesta** e quindi una **parte della memoria rimane inutilizzata**.

## Buddy system e allocatori potenza del 2

Il buddy system e gli allocatori potenza del 2 eseguono l’allocazione della memoria in blocchi di poche dimensioni predefinite. Questa caratteristica porta a **frammentazione interna** in quanto parte della memoria di ogni blocco allocato può essere sprecata. Tuttavia consente all’allocatore di mantenere **free list separate** per blocchi di dimensioni differenti. Tale organizzazione evita ricerche costose nella free list e porta ad **allocazioni e deallocazioni veloci**. Inoltre bisogna effettuare una distinzione, in quanto il **buddy** system esegue un’**unione ristretta**, mentre l’**allocatore potenza del 2** **non** **esegue** alcun tipo di **unione**. Un **buddy system** **divide e ricombina i blocchi** di memoria **in modo predefinito** durante le allocazioni e le deallocazioni. I blocchi creati dividendo un blocco prendono il nome di **blocchi** **buddy**. I blocchi buddy liberi vengono poi riuniti per formare il blocco da cui sono stati generati: quest’operazione prende il nome di **coalescenza**, o **fusione**. I blocchi liberi adiacenti che non sono buddy, non sono sottoposti a coalescenza. Ad esempio, un **buddy system binario**, divide un blocco in due parti (buddy) di pari dimensioni. Le dimensioni dei blocchi sono per differenti valori di , con una soglia minima prefissata. Tramite questo vincolo ci si assicura che i blocchi di memoria non siano inutilmente piccoli.

Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamente

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamenteL’allocatore buddy system associa un **tag** di 1 bit ad ogni blocco per indicare se il blocco è **libero o allocato**, con il tag che può trovarsi sia all’interno che all’esterno del blocco. L’allocatore mantiene **diverse liste** di blocchi liberi, **ognuna doppiamente linkata** e **formata da blocchi di stessa dimensione**, ad esempio , per qualche . L’allocatore comincia con un singolo blocco libero di dimensioni , con , che viene aggiunto alla free list di taglia . Supponiamo ora che un processo richieda un’area di memoria di byte e che un sistema trovi la più piccola potenza di 2 maggiore di , con . Se la lista di blocchi di dimensione non è vuota, allora alloca il primo blocco presente nella lista, cambiando il tag da libero ad allocato. Se invece la lista di taglia è vuota, allora si cerca la lista di blocchi di dimensione . A questo punto viene prelevato un blocco da tale lista, lo si divide in due metà di dimensione e si mette uno di questi due blocchi nella free list per blocchi di taglia , utilizzando l’altro per soddisfare la richiesta. Quando invece un processo libera un blocco di memoria di taglia , il sistema cambia il tag in libero e controlla il tag del suo blocco buddy per vedere se è libero. Nel caso in cui lo sia, i due blocchi vengono fusi in un blocco singolo di taglia . Questo controllo viene ripetuto, ossia se il buddy di questo nuovo blocco di taglia è libero allora si continuano ad effettuare queste unioni. Un blocco viene posto in una free list solo quando trova che il suo blocco buddy non è libero.

Nell’**allocatore potenza del 2** invece, la **dimensione** **dei blocchi** di memoria è formata da **potenze di** **2**. Anche in questo caso vengono mantenute free list separate per blocchi di dimensione diversa, ma in questo caso ogni blocco contiene un **header** che a sua volta contiene l’indirizzo della free list a cui dovrebbe essere aggiunto quando diventa libero. Quando viene eseguita una richiesta per byte, l’allocatore dapprima controlla la free list contenente i blocchi di dimensione , per il più piccolo valore di tale che , e nel caso sia vuota controlla la free list contenente blocchi con dimensione potenza del 2 successiva. Nell’allocatore potenza di 2 **non** **avvengono suddivisioni di blocchi**, quindi ad una richiesta viene allocato un intero blocco. Di conseguenza non viene eseguita la fusione di blocchi contigui per creare blocchi di taglia maggiore, e quindi quando un blocco viene rilasciato, torna alla sua free list.

### Indirizzo logico, indirizzo fisico e traduzione di indirizzi

Un indirizzo logico è l’indirizzo di un’istruzione o di un dato utilizzato in un processo. L’insieme degli indirizzi logici di tale processo costituisce il suo spazio di indirizzamento logico. Un indirizzo fisico invece è l’indirizzo di memoria dove è memorizzata un’istruzione o un dato. L’insieme degli indirizzi fisici nel sistema costituisce il suo spazio di indirizzamento fisico. Prendendo come esempio l’immagine precedente:

Immagine che contiene testo, diagramma, Piano, schermata

Descrizione generata automaticamente

Lo spazio di indirizzamento logico del processo P si estende da 0 a 140K-1, mentre lo spazio di indirizzamento fisico si estende da 0 a 640K-1. L’area dati xyz nel programma del processo P ha indirizzo 51488, che è il suo indirizzo logico. La componente P-1 del processo ha una dimensione di 50 KB, ossia 51200 byte, per cui xyz si trova nella componente P-2 ed ha il byte numero 288, visto che 51488 - 51200 = 288. Poichè P-2 è caricato nell’area di memoria con indirizzo di avvio 300 KB, ovvero 307200, l’indirizzo fisico di xyz è 307488. Il kernel memorizza le informazioni relative alle aree di memoria allocate ad un processo in una tabella e le rende disponibili alla MMU. La CPU invia gli indirizzi logici di ogni dato o istruzione utilizzato nel processo alla **MMU**, che a sua volta utilizza le informazioni relative all’allocazione della memoria per calcolare i corrispondenti indirizzi fisici. Questa procedura per determinare l’indirizzo fisico, ossia l’indirizzo di memoria effettivo di un dato o di un’istruzione, prende il nome di **traduzione dell’indirizzo**. Un **indirizzo logico** è formato da **due parti**: l’**id componente del processo** che contiene l’indirizzo e l’**id del byte in tale componente** dello stesso processo. L’indirizzo logico viene quindi visto come una coppia . L’**indirizzo fisico** viene poi determinato, dalla MMU, mediante la **formula**:



Se guardiamo l’esempio precedente, l’indirizzo fisico veniva proprio calcolato sommando a 307200, che era l’indirizzo di avvio della componente P-2, il numero 288, ossia il numero di byte di cui spostarci a partire dall’indirizzo di avvio.

## Paginazione

Nella vista logica lo spazio di indirizzamento di un processo consiste in un’organizzazione lineare delle pagine. Ogni pagina è composta da byte, con potenza di 2, il cui valore (di ) è specificato nell’architettura del sistema. I processi utilizzano indirizzi logici numerici e la MMU li scompone nella **coppia** , con e . Ad **esempio** se consideriamo due processi P e R in un sistema che utilizza una pagina di dimensione 1 KB, i byte sono numerati da 0 a 1023, e di conseguenza il processo P ha indirizzo di avvio 0 e dimensione 5500 byte, con 6 pagine numerate da 0 a 5. Bisogna notare inoltre come l’ultima pagina contenga solo 380 byte. Se un dato chiamato sample ha indirizzo 5248, ovvero 5 x 1024 + 128, la MMU vede il suo indirizzo come la coppia (5,128). Il processo R ha invece dimensione 2500 byte, quindi 3 pagine, numerate da 0 a 2.

Immagine che contiene diagramma, linea, schermata, Parallelo

Descrizione generata automaticamente

L’hardware partiziona la memoria in aree chiamate **frame**, numerati (i frame) a partire da 0, e che hanno la stessa **dimensione di una pagina**. Il kernel mantiene una lista, chiamata **lista dei frame liberi**, per tenere traccia dei numeri di frame liberi. Quando carica un processo, il kernel consulta la lista dei frame liberi e alloca un frame libero ad ogni pagina del processo. Per facilitare la traduzione dell’indirizzo, il kernel costruisce una **tabella delle pagine** (**PT**) **per ogni processo**. La PT ha **un elemento per ogni pagina del processo**, **che indica il frame** **allocato alla pagina**. Durante la **traduzione di un indirizzo logico** , la **MMU** **utilizza il numero di pagina** per indicizzare la PT del processo, dopodichè **ottiene il numero del frame** allocato a e **calcola l’indirizzo di memoria effettivo tramite l’equazione** vista precedentemente. Facciamo ora un **esempio** a partire dai processi P e R visti in precedenza.

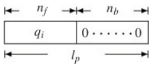


Ogni frame ha dimensione 1 KB ed il computer ha una memoria di 10 KB, di conseguenza i frame sono numerati da 0 a 9. Di questi 10, 6 sono occupati da P e 3 da R, con il frame 4 che è l’unico libero. Come detto in precedenza, la variabile sample del processo P ha indirizzo logico (5,128), quindi durante la sua esecuzione il processo la traduce nell’indirizzo effettivo di memoria seguendo l’equazione vista in precedenza. L’indirizzo di memoria effettivo di (5,128) sarà uguale all’indirizzo di avvio del frame #8 + 128, ossia 8 x 1024 + 128 = 8320. Per descrivere la traduzione dell’indirizzo mediante paginazione utilizziamo la seguente **notazione**:

* , dimensione di una pagina;
* , lunghezza di un indirizzo logico (numero di bit);
* , lunghezza di un indirizzo fisico;
* , numero di bit utilizzato per rappresentare il numero del byte in un indirizzo logico
* , numero di bit utilizzato per rappresentare il numero di pagina in un indirizzo logico;
* , numero di bit utilizzato per rappresentare il numero di frame in un indirizzo fisico.

Ricordiamo che la dimensione di una pagina è una potenza di 2, con scelto in modo che . Quindi gli bit meno significativi in un indirizzo logico danno , il byte interno della pagina, mentre i restanti bit a partire da rappresentano il numero di pagina. La MMU ottiene il valore di e semplicemente raggruppando i bit di un indirizzo logico nel seguente modo

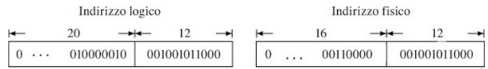
,

dove . L’uso di una potenza di 2 come dimensione della pagina semplifica la costruzione dell’indirizzo di memoria effettivo. Consideriamo adesso che la pagina sia allocata nel frame : poichè le pagine e i frame hanno le stesse dimensioni, sono necessari bit per indirizzare i byte in un frame. Di conseguenza l’indirizzo fisico del byte 0 di un frame è

dove . L’indirizzo fisico del byte nel frame è quindi dato da

.

La MMU ottiene questo indirizzo semplicemente concatenando e per ottenere un numero di bit . Andiamo ora a visionare un **esempio** vero e proprio di traduzione dell’indirizzo nella paginazione. Supponiamo di avere un computer che utilizza indirizzi logici a 32-bit e una dimensione di pagina di 4 KB e si supponga che 12 bit siano sufficienti per indirizzare i byte in una pagina. In questo modo, i 20 bit più significativi in un indirizzo logico rappresentano , mentre i 12 meno significativi rappresentano . Per una memoria di 256 MB , in questo modo i 16 bit più significativi in un indirizzo fisico rappresentano . Se la pagina 130 si trova nel frame 48, allora e , e se , gli indirizzi appaiono come segue:

.

Durante la traduzione dell’indirizzo, la MMU ottiene e semplicemente raggruppando i bit dell’indirizzo logico come mostrato sopra. Si accede quindi al 130esimo elemento della PT per ottenere , ovvero 48. Questo numero viene concatenato a per formare l’indirizzo fisico.

## Immagine che contiene testo, schermata, ricevuta, Carattere Descrizione generata automaticamenteTabella delle pagine

## Page fault

Durante la traduzione degli indirizzi la MMU controlla, nell’elemento della PT relativo alla pagina specificata nell’indirizzo logico, il valore del **bit di validità**. Nel caso in cui il bit indichi che la pagina non è presente in memoria, la MMU genera un **interrupt di page fault**. Viene quindi attivata l’ISR per verificare che l’interrupt è stato causato da un page fault, e di conseguenza invoca il gestore della memoria virtuale con il numero di pagina che ha provocato il page fault. Il gestore, a questo punto, controlla il campo “**altre info**” dell’elemento della PT relativo alla pagina che ha causato il page fault, ottenendo l’indirizzo del blocco del disco nello spazio di swap relativo al processo che contiene la pagina. Con l’indirizzo appena ottenuto il gestore consulta la lista dei frame liberi per allocarne uno alla pagina che ha provocato il page fault, avviando un’**operazione di I/O** per caricare la pagina in memoria. Quando l’operazione di I/O termina, il gestore aggiorna l’entrata (l’elemento della PT relativo alla pagina) relativa alla pagina nella PT, settando il bit di validità a 1 e ponendo il numero del frame a cui abbiamo allocato la pagina nel campo apposito. MMU e gestore della memoria virtuale, quindi, interagiscono per decidere quando una pagina di un processo deve essere caricata in memoria.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, diagramma, Disegno tecnico

Descrizione generata automaticamente

### Sostituzione delle pagine

Quando si verifica un page fault, la pagina richiesta viene caricata in un frame libero, ma nel caso in cui non ci siano frame liberi, il gestore della memoria virtuale esegue un’operazione di sostituzione della pagina. In particolare viene utilizzato un **algoritmo di sostituzione della pagina** per selezionare una delle pagine attualmente in memoria per sostituirla. Si accede all’entrata della PT relativa alla pagina selezionata per porre il bit di validità a 0, e inizia un’operazione di page-out nel caso in cui il bit “modificato” della PT indichi che è una pagina **dirty** (**bit modificato** = 1). Questo perchè nella paginazione su richiesta (paginazione della memoria virtuale credo), una copia dell’intero spazio di indirizzamento logico di un processo è mantenuta su disco, quindi nel caso in cui la pagina non venga modificata non c’è bisogno di effettuare un page-out, ma viene direttamente sostituita la pagina. Al passo successivo il gestore della memoria virtuale inizia un’operazione di page-in per caricare la pagina richiesta nel frame occupato dalla pagina su cui stiamo effettuando l’operazione di page-out. Al termine dell’operazione di page-in viene aggiornata l’entrata della pagina nella PT per registrare il numero di frame, viene posto il bit di validità ad 1 e si predispone il processo al ripristino dell’esecuzione. Durante queste operazioni il processo che ha provocato il page fault viene messo nello **stato blocked** fino a quando la pagina non viene caricata in memoria. Le operazioni di page-in e page-out richieste per implementare la paginazione su richiesta costituiscono l’**I/O di pagina**, o **traffico di pagina**, ossia l’insieme dei movimenti delle pagine da e verso la memoria. Durante la sostituzione delle pagine è importante non sostituire una pagina a cui verrà fatto riferimento nell’immediatezza. Per conoscere quali pagine non saranno riferite nel breve, il gestore della memoria virtuale utilizza **la legge empirica della località dei riferimenti**, che afferma che gli indirizzi logici usati dai processi, nel breve, tendono a raggrupparsi in specifiche porzioni del loro spazio di indirizzamento logico. Questo poichè l’esecuzione delle istruzioni è generalmente sequenziale, infatti solo il 10-20% delle istruzioni include un salto, ma anche perchè i processi tendono ad eseguire operazioni simili sugli elementi di dati non scalari (ad es. array). Per questi due motivi, i riferimenti a istruzioni e dati tendono ad essere in prossimità dei riferimenti a istruzioni e dati fatti in precedenza. Viene definita **località corrente di un processo** l’insieme delle pagine riferite in poche istruzioni precedenti. Il **principio di località dei riferimenti** indica, quindi, che l’indirizzo logico utilizzato in un’istruzione si riferisce, probabilmente, ad una pagina che si trova nella località corrente del processo. Di conseguenza, il gestore della memoria virtuale si assicura che le pagine che si trovano nella località corrente di un processo siano presenti in memoria. Questo permette un **abbassamento del numero di page fault**, che però restano comunque possibili. Per dimostrarlo, definiamo regione di prossimità di un indirizzo logico l’insieme di tutti gli indirizzi logici che sono in prossimità di esso. Si possono verificare page fault nel caso in cui la regione di prossimità non entrasse in una pagina: ciò implicherebbe che l’indirizzo successivo potrebbe essere in una pagina adiacente non inclusa nella località corrente del processo. Un’altra possibilità consiste nel fatto che un’istruzione o un dato riferito da un processo può non essere in prossimità dei precedenti riferimenti. Questa situazione prende il nome di **shift della località** di un processo.

Immagine che contiene schermata, testo, diagramma, linea

Descrizione generata automaticamente

Nella figura soprastante i cerchietti indicano i pochi indirizzi logici utilizzati durante l’esecuzione di un processo. I riquadri mostrano, invece, le regioni di prossimità di questi indirizzi, che possono anche toccare più di una pagina. La località corrente è l’insieme delle pagine marcate dal simbolo \* vicino al numero.

## Tempo effettivo di accesso alla memoria (EAT)

Il tempo effettivo di accesso alla memoria per un processo nella paginazione su richiesta è

,

con che rappresenta la **probabilità che una pagina esista in memoria** (**hit ratio**), che è il **tempo di accesso alla memoria** e che equivale all’**overhead di tempo causato dalla gestione del page fault.** Il tempo effettivo di accesso alla memoria può chiaramente essere migliorato riducendo il numero di page fault. Un modo per farlo consiste nel caricare le pagine prima che siano necessarie ad un processo. Windows, ad esempio, all’occorrenza di un page fault, carica sia la pagina che ha provocato il page fault che alcune delle pagine adiacenti. Linux, invece, consente ad un processo di specificare quali pagine dovrebbero essere precaricate.

## Allocazione della memoria ad un processo e thrashing

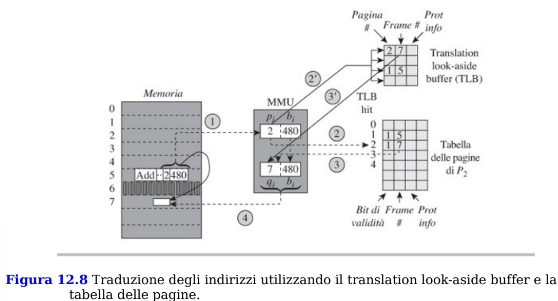
La percentuale dei page fault è maggiore se viene allocata una piccola quantità di memoria ad un processo, di conseguenza può essere diminuita aumentando la memoria allocata. Bisogna trovare però un **buon bilanciamento tra percentuale di page fault e quantità di memoria allocata**, in quanto se allocassimo tanta memoria per ridurre il page fault ma entrassero pochi processi in memoria, la CPU resterebbe spesso inattiva (idle) causando un peggioramento nelle prestazioni del sistema.

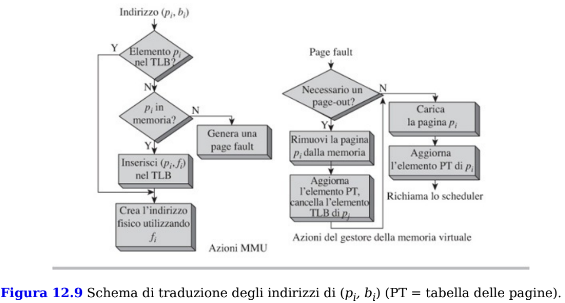
Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

Il caso peggiore di bilanciamento è quello in cui abbiamo un processo in esecuzione nella regione a bassa allocazione di memoria e ad alta frequenza di page fault. In questo caso, a causa dell’alta frequenza di page fault, il processo passerebbe molto tempo in **stato** **blocked**, e di conseguenza utilizzerebbe la CPU in maniera inefficace. Ciò causa un **elevato overhead**, dovuto anche alla commutazione di processi causata dai page fault e quindi da un elevato traffico di pagine. Questa condizione, in cui coincidono un **elevato traffico di pagine** ed una **scarsa efficienza di CPU**, prende il nome di **thrashing**.

## Traduzione degli indirizzi e generazione del page fault (TLB)

Per eseguire la traduzione degli indirizzi la MMU segue i passi della **formula EAT**. Un riferimento alla PT durante la traduzione degli indirizzi utilizza un ciclo di memoria poichè la PT si trova in memoria. Per velocizzare la traduzione dell’indirizzo si utilizza il **Translation look-aside buffer** (**TLB**), una memoria associativa piccola e veloce, che permette di eliminare il riferimento alla PT. Il TLB contiene **entrate nella forma** **(#pagina, #frame, prot\_info)** per poche pagine, a cui si è acceduto di recente e che sono in memoria.

Quando si vanno a controllare le entrate nel TLB si sta effettuando un’operazione di **TLB lookup**. Durante questa operazione, se la pagina richiesta si trova nel TLB ed ha lo stesso numero della PT allora si ottiene un “**hit**”, quindi la MMU utilizza il frame dell’entrata per completare la traduzione. Nel caso in cui invece la pagina richiesta non combaci con il suo numero nella PT si verifica un “**miss**”, e quindi la MMU accede direttamente alla PT per completare la traduzione, nel caso in cui la pagina sia in memoria. Se invece la pagina non è presente neanche nella PT significa che deve essere caricata dal gestore della memoria virtuale dallo spazio di swap del disco in memoria.

Nuove entrate nel TLB possono essere inserite sia dall’hardware che dal gestore della memoria virtuale, con la prima che risulta più efficiente. L’hardware può infatti creare una nuova entrata nel TLB ogni volta che deve completare una traduzione di indirizzi tramite un riferimento alla PT. Quando il TLB è gestito dal gestore della memoria virtuale, invece, la MMU genera un interrupt “**missing TLB entry**” ogni volta che non riesce a trovare un’entrata per la pagina richiesta nel TLB. Per creare l’entrata nel TLB per la pagina richiesta, il gestore della memoria virtuale deve eseguire diverse istruzioni. Secondo questo approccio, in pratica, la MMU esegue traduzione di indirizzi esclusivamente tramite il TLB, mentre il gestore della memoria virtuale utilizza la PT. Questa soluzione consente maggiore flessibilità poichè il gestore della memoria virtuale può utilizzare diverse organizzazione della PT per risparmiare memoria. Alcune caratteristiche sono comuni ad entrambi gli approcci. Viene infatti usato un algoritmo di sostituzione per decidere quale entrata nel TLB dovrebbe essere sovrascritta con una nuova entrata. Inoltre, l’uso del TLB può minare la protezione se la MMU esegue traduzione di indirizzi tramite le entrate nel TLB create durante l’esecuzione di qualche altro processo. Per evitare questo problema, **si aggiunge un campo all’entrata del TLB** che specifica l’**id del processo** nella quale (l’entrata) è stata creata, in modo che la MMU non la utilizzi quando qualche altro processo è in esecuzione. In alternativa, il kernel deve ripulire (**flush**) il TLB mentre esegue la commutazione tra processi.

## Organizzazione pratica della PT

Un processo con un ampio spazio di indirizzamento richiede una PT che occupi molta memoria. Per ridurre la dimensione della memoria assegnata alle PT vengono seguiti due approcci:

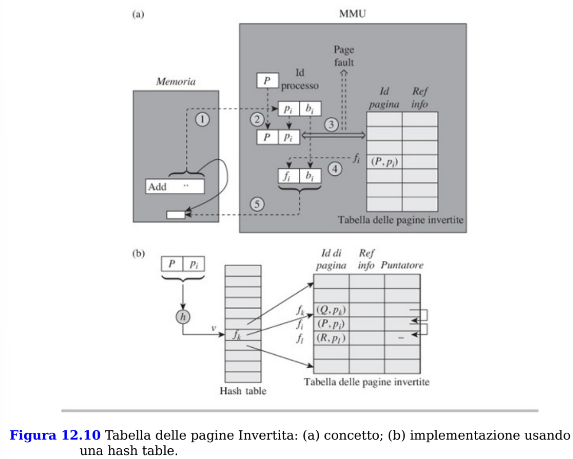
* **tabella delle pagine invertita** (**IPT**), che ha **un’entrata per ogni frame che indica quale pagina**, se esiste, **occupa il frame** (in pratica è il contrario di una PT normale). La dimensione di una IPT dipende dalla dimensione della memoria; è infatti indipendente dal numero e dalla dimensione dei processi. Con questo approccio, quindi, non si può accedere direttamente alla pagina come nella PT;
* **tabella delle pagine multilivello**, che è **essa stessa paginata**, di conseguenza non ha bisogno di essere sempre conservata in memoria. Viene utilizzata una **PT di alto livello** per accedere alle pagine della PT. Se la PT di alto livello è grande, allora anch’essa può essere paginata e così via. In questo approccio, l’accesso alle entrate della PT di una pagina deve essere effettuato attraverso le entrate delle PT di alto livello.

### 

### Tabella delle pagine invertite (IPT)

Ogni entrata della IPT è una **coppia** ordinata formata da un id**entificativo di processo e dal numero di una pagina** . Un’entrata nella IPT significa quindi che il frame è occupato dalla pagina di un processo . Quando un processo viene schedulato, lo scheduler copia l’id del processo, che chiamiamo , dal suo PCB in un registro della MMU. La MMU esegue quindi la **traduzione degli indirizzi** a partire da un indirizzo logico in un processo, mediante i **seguenti passi**:

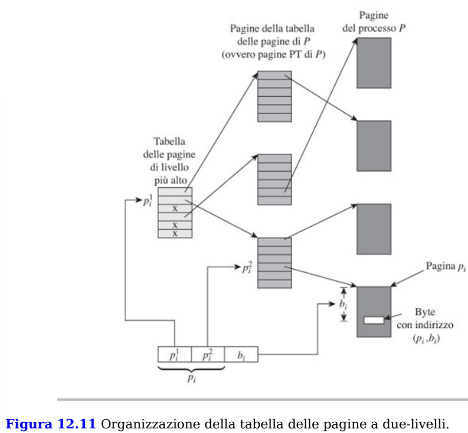
1. separare le componenti e dell’indirizzo logico;
2. formare la coppia ) con l’id del processo ;
3. cercare questa coppia nella IPT, generando un page fault nel caso in cui non esista nella IPT;
4. se la coppia ) esiste nell’entrata del frame della IPT, bisogna copiare il numero del frame per utilizzarlo nella traduzione degli indirizzi;
5. calcolare l’indirizzo effettivo di memoria utilizzando e .



Per velocizzare la ricerca della coppia ) viene utilizzata una **hash table**. In pratica ogni entrata della IPT contiene un ulteriore **campo puntatore** **che punta** **ad un’altra entrata nella stessa tabella**, questo poichè la dimensione della hash table è ovviamente molto più piccola rispetto al numero totale di pagine di tutti i processi in memoria. Infatti la **funzione di hash** viene calcolata prendendo il resto della divisione tra una stringa di bit che rappresenta la coppia ) concatenata, e poi interpretata come numero intero, e la dimensione della hash table.

### Tabella delle pagine multilivello

L’idea della tabella delle pagine multilivello è quella di **paginare la PT** **e** di **caricare** **le sue pagine su richiesta**. Questo approccio richiede un **indirizzamento a due livelli**, in cui una **PT di alto livello** contiene le entrate che riportano le informazioni riguardanti le pagine della PT, mentre la PT contiene le informazioni riguardanti le pagine del processo. La memoria, quindi, contiene due tipi di pagine: le **pagine dei processi** e le **pagine delle PT dei processi**, che chiamiamo **pagine PT**. Per la **traduzione degli indirizzi** è **necessario che siano in memoria sia la pagina del processo che la pagina PT**.



Se la dimensione di un’entrata della tabella è byte, il numero di entrate della tabella in una pagina PT è . L’**indirizzo logico** è raggruppato **in tre campi**:

.

La pagina PT di numero contiene l’entrata della PT per , mentre è il numero dell’entrata per nella pagina PT.

## Politiche di sostituzione delle pagine

L’obiettivo delle politiche di sostituzione delle pagine è quello di sostituire una pagina che probabilmente non sarà referenziata nell’immediato futuro. Esistono **tre principali politiche**:

* politica di **sostituzione ottimale** delle pagine;
* politica di **sostituzione FIFO**;
* politica di **sostituzione LRU** (**Least Recently Used**).

Per analizzare queste politiche ci si basa sul concetto di **stringa dei riferimenti alle pagine**, ossia una sequenza di pagine a cui un processo ha fatto accesso durante la sua esecuzione. Per convenienza, ad ogni stringa dei riferimenti alle pagine viene associata una **stringa dei tempi dei riferimenti**, che corrispondono, in ordine, alla stringa dei riferimenti alle pagine.

### Sostituzione ottimale delle pagine

La sostituzione ottimale consiste nel prendere decisioni per la sostituzione di pagine in modo che il **page fault** durante l’esecuzione di un processo sia il **più** **piccolo possibile**. Per realizzare la sostituzione ottimale, ad ogni page fault, la politica dovrebbe considerare tutte le possibili decisioni alternative, analizzare le loro implicazioni su page fault futuri e selezionare la migliore alternativa. Questa **politica** è chiaramente **inapplicabile** in quanto il gestore della memoria virtuale non può conoscere il comportamento futuro di un processo. Come strumento analitico, però, questa politica fornisce un utile confronto in relazione alle prestazioni delle altre politiche di sostituzione. Sebbene la sostituzione ottimale sembri richiedere un’eccessiva analisi, Belady mostrò che **è** **equivalente** alla semplice regola per cui al verificarsi di un page fault, bisogna **sostituire la pagina il cui successivo riferimento è il più lontano nella stringa dei** **riferimenti alle pagine**.

### Sostituzione delle pagine FIFO

Nella sostituzione FIFO, ad ogni page fault viene sostituita la **pagina che è stata caricata in memoria prima di ogni altra pagina del processo**. Per semplificare il lavoro, il gestore della memoria virtuale memorizza nel campo “**ref** **info**” delle entrate nella PT l’**istante di caricamento di una pagina**.

### Sostituzione delle pagine LRU (Least Recently Used)

La politica LRU utilizza il principio di località dei riferimenti come base delle sue decisioni per la sostituzione. Ad ogni page fault viene sostituita la **pagina utilizzata meno recentemente** con la pagina richiesta. L’entrata della PT memorizza l’**istante dell’ultimo riferimento alla pagina**; quest’informazione viene chiaramente inizializzata quando la pagina è caricata, e viene aggiornata ogni volta che la pagina è riferita.

## Proprietà dello stack

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamentePer raggiungere la **situazione ottimale relativamente ai page fault**, ossia

una politica di sostituzione di pagina deve possedere la **proprietà dello stack**, anche detta **proprietà di inclusione**.

Immagine che contiene testo, Carattere, bianco, ricevuta

Descrizione generata automaticamente

con che indica l’insieme delle pagine in memoria all’istante di tempo se per tutta l’esecuzione del processo ( è un istante successivo a ma precedente a ). Per spiegare meglio la proprietà dello stack consideriamo due esecuzioni del processo , con e , rispettivamente, dove . Se una politica ha la proprietà dello stack, allora negli stessi istanti, durante le operazioni di nelle due esecuzioni, tutte le pagine che erano in memoria con sarebbero in memoria anche quando . Inoltre, la memoria contiene anche pagine aggiuntive del processo. Se una di tali pagine sarà riferita in pochi riferimenti successivi di , il page fault si verifica se ma non se . Di conseguenza il page fault è più elevato se rispetto a se . La **politica FIFO**, però, **non rispetta** sempre **la proprietà dello stack**, a differenza della **politica LRU** che **rispetta** **la** **proprietà** in ogni istante.

Immagine che contiene testo, ricevuta, diagramma, linea

Descrizione generata automaticamente

Possiamo notare da questo grafico che, nella politica FIFO, all’aumentare dell’allocazione di memoria per il processo corrisponde un aumento nel numero di page fault. Quest’anomalia prende il nome di **anomalia di Belady**, dal nome di colui che per primo riportò questo comportamento. Il gestore della memoria virtuale, quindi, non può utilizzare una politica FIFO, in quanto l’anomalia appena vista renderebbe difficile affrontare il thrashing del sistema. Al contrario, il gestore della memoria virtuale può utilizzare la politica LRU poichè questa è una funzione non crescente, e quindi è possibile combattere il thrashing allocando più memoria.

## Politiche di sostituzione delle pagine in pratica (algoritmi di sostituzione)

Il gestore della memoria virtuale mantiene una **lista di frame liberi**, provando a tenere pochi frame in questa lista ad ogni istante. Il gestore consiste di **due thread** **demoni**:

* il **gestore dei frame liberi**, che viene attivato dal gestore della memoria virtuale quando il numero di frame liberi si trova al di sotto di una soglia predefinita dallo stesso gestore (della memoria virtuale). A questo punto il gestore dei frame liberi scandisce le pagine presenti in memoria per identificare alcune pagine da poter liberare, per poi aggiungere i frame occupati da queste pagine alla lista dei frame liberi. Questo thread demone, tra l’altro, è quello **deputato** ad occuparsi effettivamente dell’implementazione della **politica di sostituzione delle pagine**;
* Immagine che contiene testo, schermata, diagramma, Carattere

  Descrizione generata automaticamenteil **gestore dell’I/O di pagina**, il quale esegue le operazioni di page-in e page-out.

Dalla figura soprastante vediamo il funzionamento effettivo della sostituzione delle pagine. Notiamo anche un ulteriore gestore, oltre a quelli già visti, ossia il **gestore** **dei page fault**, che funziona come il gestore degli eventi del kernel: **viene** infatti **attivato quando si verifica un page fault**. La **politica di sostituzione LRU** sarebbe la scelta automatica come politica di sostituzione delle pagine, tuttavia, **non** è **applicabile** perchè i **computer non hanno sufficienti bit nel campo** “**ref info**” per memorizzare l’istante di ultimo riferimento; utilizzano infatti un solo bit, ossia il bit di riferimento. Potendo utilizzare il solo **bit di riferimento**, è stata creata, per esigenza, una nuova classe di politiche chiamate **politiche Not Recently Used** (**NRU**), in cui il bit di riferimento viene utilizzato per determinare se una pagina è stata riferita di recente. Una semplice politica NRU inizializza il bit di riferimento a 0 quando una pagina viene caricata, mentre lo imposta ad 1 quando la pagina viene riferita. Quando è necessaria una sostituzione di pagina, se il gestore della memoria virtuale verifica che i **bit di riferimento di tutte le pagine** sono diventati **1**, **resetta tutti i bit a 0** e **seleziona arbitrariamente una delle pagine** per la sostituzione. Nel caso in cui ci sia almeno un bit di riferimento a 0, sostituisce la pagina con quel bit. Gli algoritmi di sostituzione delle pagine, detti **algoritmi di clock**, forniscono la miglior discriminazione tra le pagine, resettando i bit di riferimento delle pagine periodicamente, piuttosto che farlo quando tutti i bit diventano 1. In questo modo risulta possibile sapere se una pagina è stata riferita nell’immediato passato, piuttosto che rispetto a quando tutti i bit sono stati resettati. Negli algoritmi di clock, le pagine di tutti i processi sono inseriti in una **lista circolare** ed i **puntatori** utilizzati dagli algoritmi si muovono sulle pagine ripetutamente. La pagina puntata dal puntatore è quella in esame, e a seguito dell’azione su di essa, il puntatore viene aggiornato per puntare alla pagina successiva. Esistono **due tipi di algoritmi di clock**:

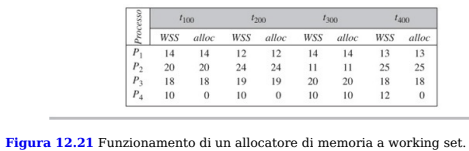
* gli **algoritmi di clock ad una lancetta**, nei quali un’**esecuzione** è composta da **due passi** su ogni pagina. In particolare, nel primo passo, il gestore della memoria virtuale resetta semplicemente il bit di riferimento della pagina puntata dal puntatore, mentre nel secondo passo cerca tutte le pagine i cui bit di riferimento sono a 0 e li aggiunge alla lista dei frame liberi;
* Immagine che contiene diagramma, schermata, cerchio, linea

  Descrizione generata automaticamentegli **algoritmi di clock a due lancette**, i quali utilizzano **due puntatori**. Uno serve a resettare i bit di riferimento, e prende il nome di **puntatore d**i **reset** (**RP**), mentre l’altro viene utilizzato per controllare i bit di riferimento, e prende il nome di **puntatore di controllo** (**EP**). Il frame puntato da EP viene aggiunto alla lista dei frame liberi se il suo bit di riferimento è 0, mentre il bit di riferimento del frame puntato da RP viene impostato a 0. E’ inoltre importante ricordare che i due puntatori sono incrementati simultaneamente.

Le **proprietà degli algoritmi di clock** sono definite dalla **distanza dei puntatori**. Se i puntatori sono vicini, infatti, sarà esaminata una pagina subito dopo il reset del suo bit di riferimento, di conseguenza rimarranno in memoria solo le pagine usate di recente. Se invece i puntatori dei clock sono distanti, solo le pagine che non vengono utilizzare da molto tempo saranno rimosse.

## Working set

Nell’allocazione di memoria variabile, quando si utilizza la sostituzione locale delle pagine (ossia quando vengono prese in considerazione per la sostituzione solo le pagine relative al processo in cui si è verificato il page fault), il gestore deve, come abbiamo già detto, determinare il giusto valore di per un processo, di volta in volta. Per farlo, può utilizzare il **modello working set**, il quale fornisce una base, appunto, per decidere quante e quali pagine di un processo dovrebbero essere mantenute in memoria per ottenere una buona prestazione di processo. Quando un gestore della memoria virtuale utilizza il modello working set, si dice che utilizzi un **allocatore di memoria working set**. Il working set è l’**insieme delle pagine di un processo** che sono state **riferite** **nelle** **precedenti** **istruzioni del processo**, dove è un parametro di sistema. Le precedenti istruzioni del processo costituiscono la **finestra del working set**. Viene utilizzata la notazione per rappresentare il working set per il processo all’istante con dimensione della finestra e la notazione per rappresentare la dimensione del working set , ossia il numero di pagine in . Bisogna notare, inoltre, che poichè una pagina può essere riferita più volte in una finestra di WS. Un allocatore di memoria basato su WS mantiene l’intero WS di un processo in memoria, oppure sospende il processo. Così, ad ogni istante di tempo , **un processo** **può avere solamente** **oppure** . Questa strategia assicura un **buon tasso di successo** (hit ratio) **in memoria**, grazie al **principio della località dei riferimenti**, ed **evita** **il** sottodimensionamento di memoria per un processo, evitando così il **thrashing**. L’allocatore di memoria working set varia il g**rado di multiprogramazione** in base ai cambiamenti nelle dimensioni dei working set dei processi. Per esempio se è l’insieme dei processi in memoria, il grado di multiprogrammazione dovrebbe essere **decrementato** se , dove è il numero totale dei frame in memoria. Per farlo, l’allocatore working set rimuove alcuni processi dalla memoria, finchè . Dovrebbe, invece, essere **incrementato** il grado di multiprogrammazione se ed esiste un processo tale che . Il gestore della memoria virtuale mantiene due elementi di informazione per ogni processo, e . Quando il grado di multiprogrammazione deve essere **ridotto**, il gestore della memoria virtuale seleziona un processo , esegue un page-out per ogni pagina modificata dallo stesso e cambia lo stato dei frame allocati ad esso a libero. Di conseguenza viene settato a 0, mentre il valore di resta immutato. Quando invece il grado di multiprogrammazione deve essere **incrementato** e il gestore della memoria virtuale decide di ripristinare un processo , setterà e allocherà un numero di frame pari al valore di . Viene quindi caricata la pagina di che contiene la prossima istruzione da eseguire e quando si verificano dei page fault, dovrebbero essere caricate altre pagine. In alternativa il gestore della memoria virtuale carica tutte le pagine di quando viene ripristinata l’esecuzione di , tuttavia questo approccio può portare a caricamenti ridondanti di pagine, poichè alcune pagine in possono essere riferite più volte. Essendo costoso determinare e ad ogni istante , il gestore della memoria virtuale che utilizza un allocatore di memoria working set può determinare i **working set periodicamente** piuttosto che appunto ad ogni istante. I working set determinati alla fine di un intervallo vengono utilizzati per decidere i valori di da usare nell’intervallo successivo.



Come possiamo ben notare dalla figura soprastante, ad ogni istante di tempo vengono ricalcolati e , con quest’ultimo che può assumere solamente i valori 0 o , in base a se ci sono abbastanza frame liberi o meno. L’esempio viene condotto considerando 60 frame in memoria.

## Copy-on-write

La funzionalità copy-on-write permette di risparmiare memoria quando i **dati** **nelle** **pagine condivise** potrebbero essere **modificati**, ma i **valori modificati** sono **riservati** **ad un solo processo**. Viene quindi utilizzato un **flag copy-on-write** nelle entrate della PT, settato a , quando il flag è impostato per quella pagina.